See all versions
of this document

Vitis Model Composer User Guide

UG1483 (v2021.2) October 22, 2021

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
10/22/2021 Version 2021.2	
AI Engine Blockset	Updated to add Mixer and DDS blocks.
AI Engine Utilities	Added sections.
Interconnecting AI Engine and HDL Blocks	Clarification to AIE-HDL/HDL-AIE interconnection.
AI Engine DSPLib	Added sections.
06/16/2021 Version 2021.1	
Document Title and Revision Summary	Changed title to ''Vitis Model Composer User Guide''
General updates	General updates for release 2021.1

Table of Contents

Overview

Navigating Content by Design Process

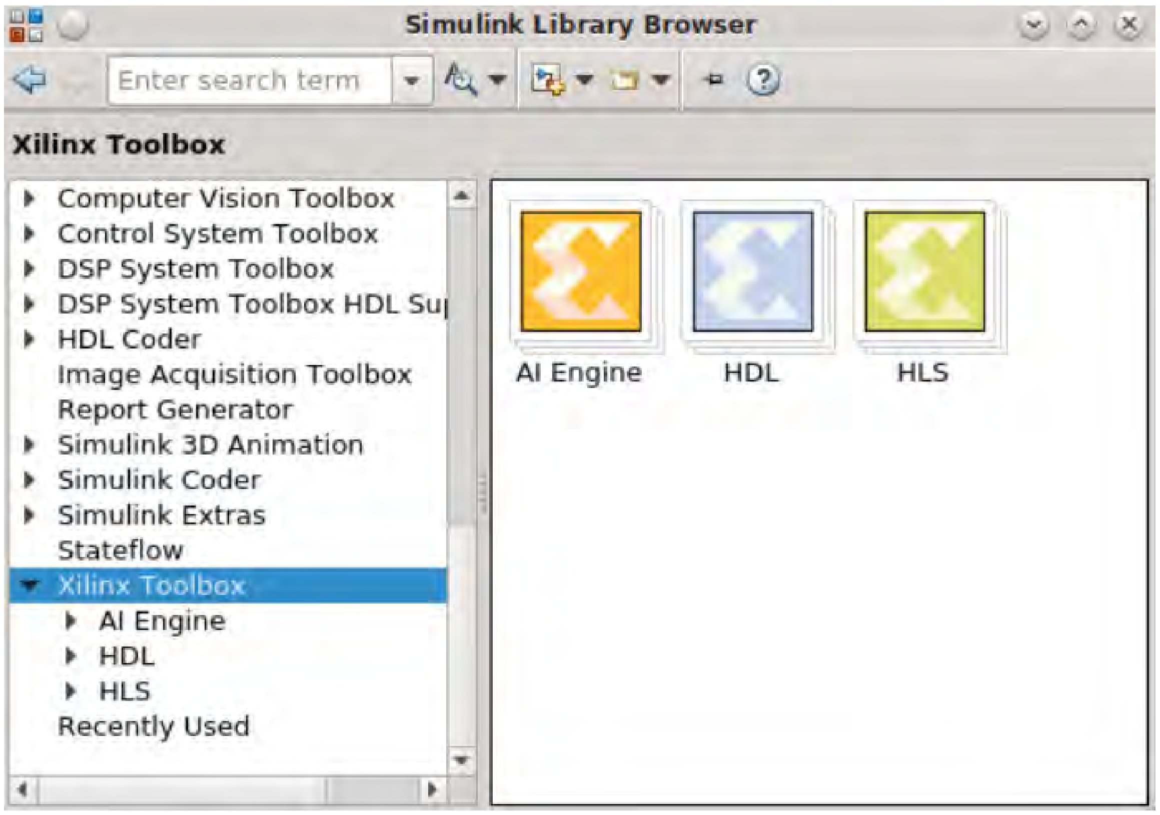
Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. All Versal™ ACAP design process Design Hubs can be found on the Xilinx.com website. This document covers the following design processes:

- **AI Engine Development:** Creating the AI Engine graph and kernels, library use, simulation debugging and profiling, and algorithm development. Also includes the integration of the PL and AI Engine kernels. Topics in this document that apply to this design process include:
  - Creating an AI Engine Design using Model Composer
  - Simulation and Code Generation
  - Chapter 5: Connecting AI Engine and Non-AI Engine Blocks
- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the Vivado® timing, resource use, and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:
  - Hardware Design Using HDL Library
  - Creating a Model Composer Design
  - Compilation Types for HDL Library designs
- **System Integration and Validation:** Integrating and validating the system functional performance, including timing, resource use, and power closure. Topics in this document that apply to this design process include:
  - Performing Analysis in Model Composer
  - Using Hardware Co-Simulation

Introduction

Vitis™ Model Composer is a model-based design tool that enables rapid design exploration within the Simulink environment and accelerates the path to production on Xilinx devices through automatic code generation. This provides a library of performance-optimized blocks for design and implementation of algorithms on Xilinx devices using HDL, HLS, and AI Engine blocks.

Figure 1: **Simulink Library Browser**



You can focus on expressing algorithms using blocks from these libraries as well as custom user-imported blocks, without worrying about implementation specifics, and leverage all the capabilities of Simulink's graphical environment for algorithm design, simulation, and verification.

The AI Engine library in Vitis Model Composer includes:

- Blocks that support connection between the AI Engine and the Xilinx HDL blockset.
- Block to import HLS kernels which can be targeted to the PL portion of Versal devices.
- Blocks to import kernels and graphs which can be targeted to the AI Engine portion of Versal™ ACAP devices.
- Configurable AI Engine functions such as FIR and FFT.

The HDL library in Vitis Model Composer contains DSP building blocks. These blocks include the basic element blocks such as adders, multipliers, and registers. Also included are a set of complex DSP building blocks such as FFTs, filters, and memories. These blocks leverage the Xilinx IP core generators to deliver optimized results for the selected device.

The HLS library in Vitis Model Composer offers predefined blocks which includes functional blocks for Math, Linear Algebra, Logic, and Bit-wise operations. The tool transforms your algorithmic specifications to production-quality implementation through automatic optimizations that extend the Xilinx High Level Synthesis technology.

The rest of this document describes information on features and specific blocks related to the:

- **HDL Library:** Refer to Chapter 2: HDL Library
- **HLS Library:** Refer to Chapter 3: HLS Library
- **AI Engine Library:** Refer to Chapter 4: AI Engine Library

What's New and Limitations

System Generator - the previous standalone design environment to develop DSP algorithms and generate HDL as an output - is now part of Vitis™ Model Composer. As a result of this product unification, HLS, AI Engine, and System Generator libraries in the Xilinx toolbox have been merged to develop algorithms in a single MATLAB session.

The HDL Library in the Xilinx Toolbox contains all the library blocks previously contained in the System Generator blockset. Any existing System Generator design can be opened in Vitis Model Composer. Furthermore, you can develop new algorithms using the HDL library similarly to using the System Generator tool.

For information related to what is new for a specific release of Model Composer, refer to *What's New* at https://www.xilinx.com/products/design-tools/vivado/integration/model-composer.html#new.

In addition, while Model Composer is a toolbox built onto the MathWorks Simulink environment, there are certain features of Simulink that are not supported in Model Composer. The following is a list of some of the unsupported features:

- Simulink Performance Advisor.
- Model referencing.
- Variant subsystems.
- Model Composer blocks do not support Simulink fixed-point types and only support Xilinx® fixed-point types.

- Fixed-point designer does not integrate with Model Composer.
- Accelerator mode and Rapid Accelerator mode.

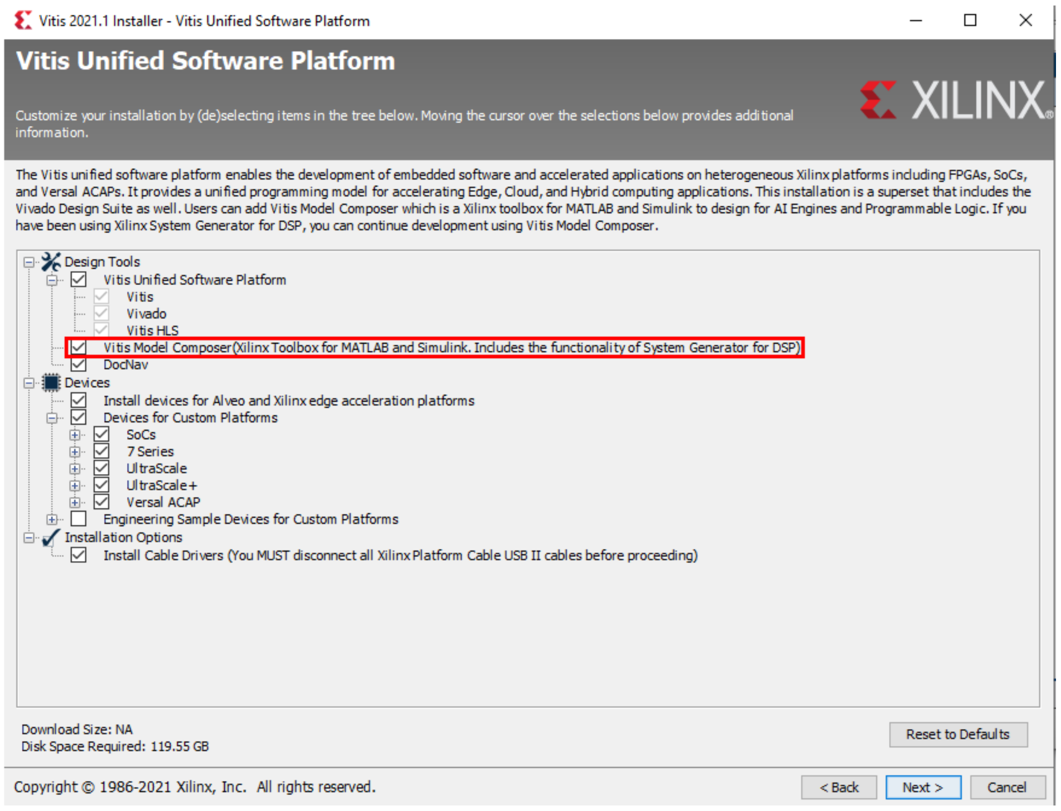
Installation

Downloading

Vitis™ Model Composer is part of Vivado® as well as the Vitis software platform, which can be downloaded from the Xilinx website. The AI Engine library is available only in the Vitis software platform, it is not part of the Vivado installation.

The Vitis Model Composer tool is selected by default in the Xilinx Unified Installer Window irrespective of whether you choose Vitis or Vivado as a product. The following figure shows the Vitis installer window and the Vitis Model Composer install option.

Figure 2: **Vitis Installer**



Vitis 2021.1 Installer - Vitis Unified Software Platform

Vitis Unified Software Platform

Customize your installation by (de)selecting items in the tree below. Moving the cursor over the selections below provides additional information.

The Vitis unified software platform enables the development of embedded software and accelerated applications on heterogeneous Xilinx platforms including FPGAs, SoCs, and Versal ACAPs. It provides a unified programming model for accelerating Edge, Cloud, and Hybrid computing applications. This installation is a superset that includes the Vivado Design Suite as well. Users can add Vitis Model Composer which is a Xilinx toolbox for MATLAB and Simulink to design for AI Engines and Programmable Logic. If you have been using Xilinx System Generator for DSP, you can continue development using Vitis Model Composer.

- Design Tools
  - ✓ Vitis Unified Software Platform
    - ✓ Vitis
    - ✓ Vivado
    - ✓ Vitis HLS
  - ✓ Vitis Model Composer(Xilinx Toolbox for MATLAB and Simulink. Includes the functionality of System Generator for DSP)
  - ✓ DocNav
- Devices
  - ✓ Install devices for Alveo and Xilinx edge acceleration platforms
  - ✓ Devices for Custom Platforms
    - ✓ SoCs
    - ✓ 7 Series
    - ✓ UltraScale
    - ✓ UltraScale+
    - ✓ Versal ACAP
  - ☐ Engineering Sample Devices for Custom Platforms
- Installation Options
  - ✓ Install Cable Drivers (You MUST disconnect all Xilinx Platform Cable USB II cables before proceeding)

Download Size: NA
Disk Space Required: 119.55 GB

Copyright © 1986-2021 Xilinx, Inc. All rights reserved.

Launching Vitis Model Composer

You can launch the Vitis Model Composer tool directly from the desktop or from the command line. Double click the **Vitis Model Composer** icon to launch it from the Start Menu in Windows, or use the following command from the command prompt.

```
model_composer
```

TIP: *The command-line use of Vitis Model Composer requires that the command shell has been configured as follows. You must change directory to `<install_dir>/Model_Composer/<version>` and run the `settings64-Model_Composer.bat` (or `.sh`) file. Where `<install_dir>` is the installation folder and `<version>` is the specific version of the tool. MATLAB opens, and the HLS, HDL and AI Engine libraries and features are overloaded onto this environment.*

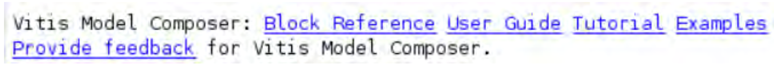
Vitis Model Composer supports the latest releases of MATLAB. For more information on Supported MATLAB versions and operating systems, refer to Supported MATLAB Versions and Operating Systems. If you have multiple versions of MATLAB installed on your system, the first version found in your PATH will be used by the tool. You can edit the PATH to move the preferred version of MATLAB to precede other versions. You can also direct the tool to open a specific version of the tool using the `-matlab` option as follows:

```
model_composer -matlab C:\Progra~1\MATLAB\R2020a
```

TIP: *When you specify the path to the MATLAB version, do not specify the full path to the executable (`bin/MATLAB`). The string `C:\Progra~1\` is a shortcut to `C:\Program Files\` which eliminates spaces from the command path. The command-line use of the Vitis Model Composer tool requires that the command shell has been properly configured as previously discussed.*

After launching the tool, you will see the following in the MATLAB command window. Use these links to access the documentation and product examples.

Figure 3: **Documentation and Examples Links**



```
Vitis Model Composer: Block Reference User Guide Tutorial Examples
Provide feedback for Vitis Model Composer.
```

Note: Model Composer simulation and code generation is not supported in Windows OS for designs that include blocks from the AI Engine library.

Supported MATLAB Versions and Operating Systems

Vitis Model Composer supports the following MATLAB versions:

- R2020a
- R2020b
- R2021a

The following operating systems are supported on x86 and x86-64 processor architectures:

- **Windows 10 Pro and Enterprise:** 10.0 1903 Update; 10.0 1909 Update; 10.0 2004 Update; 10.0 20H2; 10.0 21H1
- **Red Hat Enterprise Workstation/Server 7:** 7.8; 7.9
- **Ubuntu Linux:** 18.04.4 LTS; 18.04.5 LTS; 20.04 LTS

  ***Note*:** MATLAB 2020a version is not supported on Ubuntu 20 OS.

  - **Prerequisites for using Model Composer on Ubuntu 20 OS:**
    - QT4 library should be installed.
    - Ubuntu 20 comes with gcc 7.x to 9.x versions by default; you need to either install gcc 6.x manually or create a symbolic link using the following sudo commands:
      - `sudo ln -s /usr/include/asm-generic /usr/include/asm`
      - `sudo ln -s /usr/include/x86_64-linux-gnu/sys /usr/include/sys`
      - `sudo ln -s /usr/include/x86_64-linux-gnu/bits /usr/include/bits`
      - `sudo ln -s /usr/include/x86_64-linux-gnu/gnu /usr/include/gnu`
    - Ubuntu 20 by default comes with dash shell. To avoid any issue that you may encounter while running the downstream AI Engine flows, it is recommended to change the shell from dash to bash using the following sudo command:
      - `sudo dpkg-reconfigure dash`
- **SUSE Enterprise Linux:** 12.4, 15.2

  ***Note*:** SUSE Enterprise Linux OS is supported only for the HDL library.

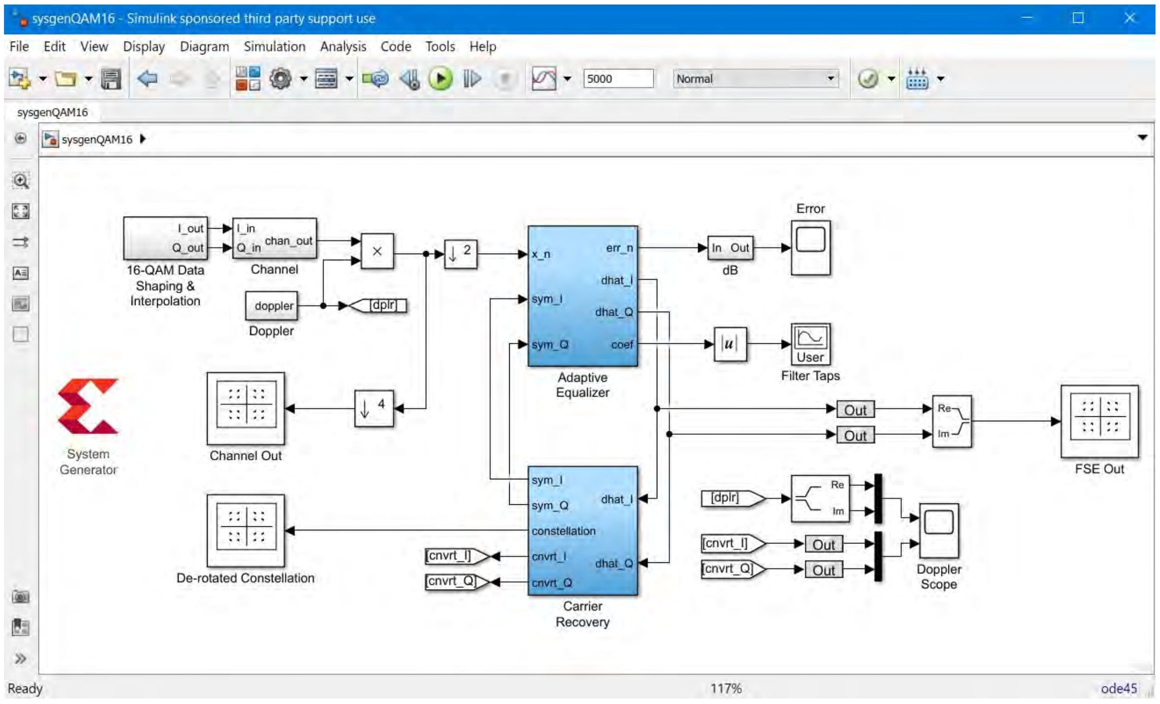
Chapter 2

HDL Library

Introduction

Vitis™ Model Composer provides the HDL blockset in Xilinx toolbox that enables the use of the MathWorks model-based Simulink design environment for FPGA design. Previous experience with Xilinx FPGAs or RTL design methodologies are not required when using the Model Composer HDL blockset. Designs are captured in the DSP friendly Simulink modeling enviornment using the HDL blockset. The Model Composer design can then be imported into a Vivado IDE project using the IP catalog.

Figure 4: **Model Composer HDL Design**

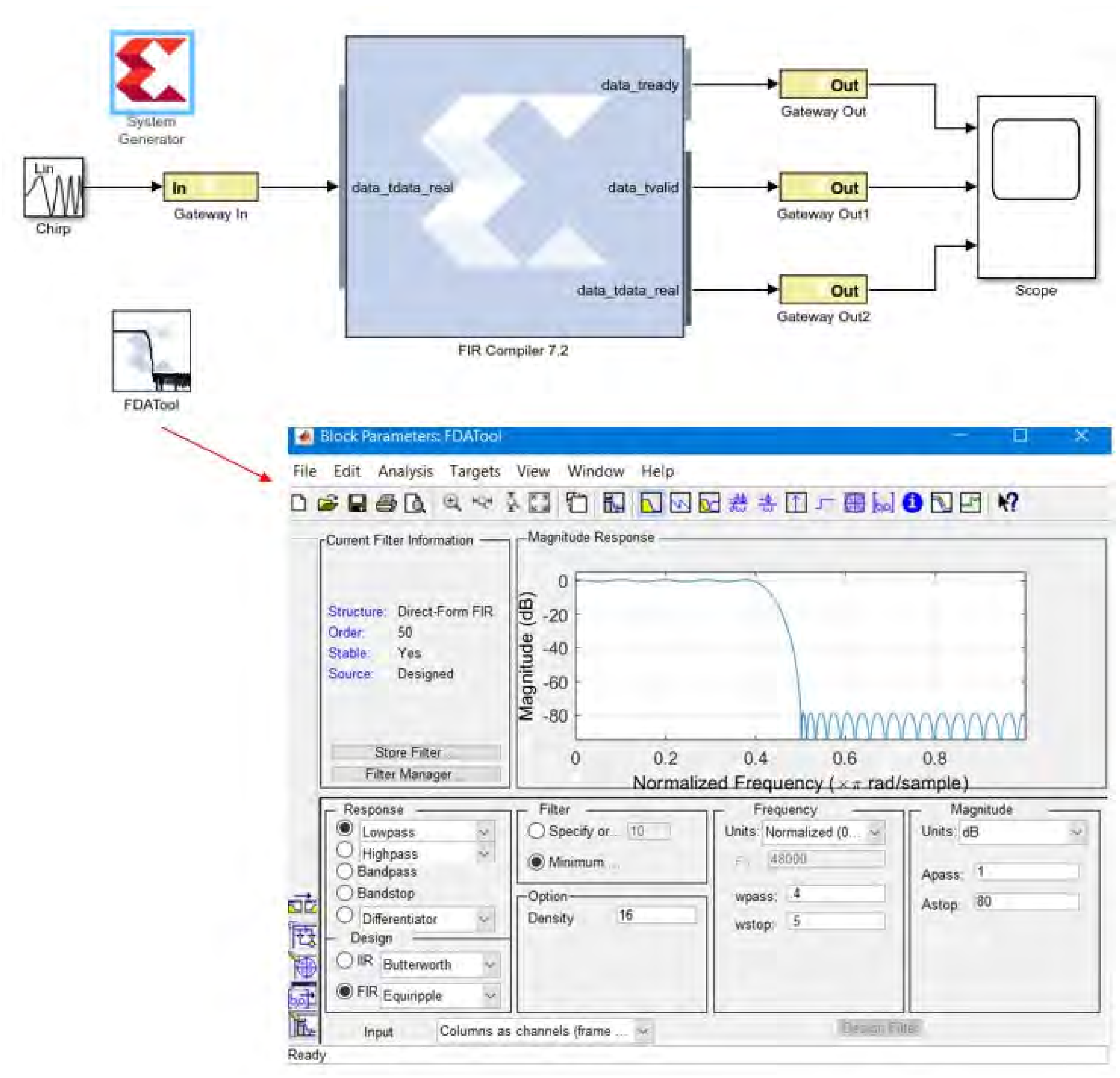


Refer to the *Vitis Model Composer Tutorial* (UG1498) for hands-on lab exercises and step-by-step instruction on how to create a model using HDL blockset and then import that model into a Vivado IDE project.

FIR Filter Generation

The HDL Blockset in Model Composer includes a FIR Compiler block that targets the dedicated DSP48E1, DSP48E2, and DSP58 hardware resources in the 7 series, UltraScale™ and Versal™ devices respectively to create highly optimized implementations. Configuration options allow generation of single rate, interpolation, decimation, Hilbert, and interpolated implementations. Standard MATLAB® functions such as fir2 or the MathWorks FDA tool can be used to create coefficients for the Xilinx® FIR Compiler.

Figure 5: **FDA Tool Example**



Structure: Direct-Form FIR
Order: 50
Stable: Yes
Source: Designed

Response: [✓ Lowpass] Highpass Bandpass Bandstop Differentiator

Design: IIR Butterworth [✓ FIR Equiripple]

Filter: Specify order: 10 [✓ Minimum order]

Option: Density 16

Frequency: Units: Normalized (0... Fs: 48000 wpass: .4 wstop: .5

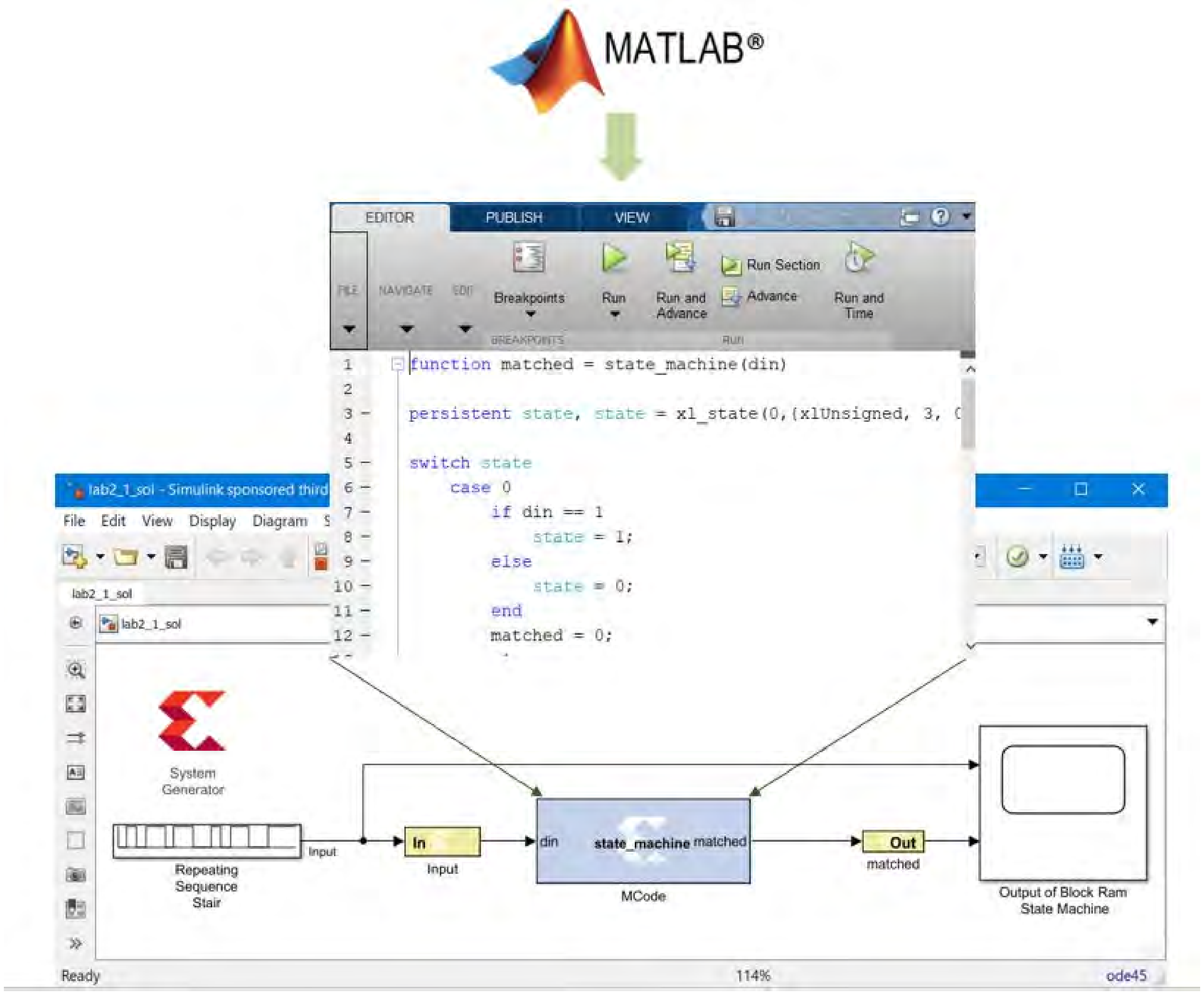
Magnitude: Units: dB Apass: 1 Astop: 80

Input: Columns as channels (frame ...

Support for MATLAB

The HDL Library in Model Composer consists of an MCode block that allows the use of non-algorithmic MATLAB® for the modeling and implementation of simple control operations.

Figure 6: **MCode Block Example**

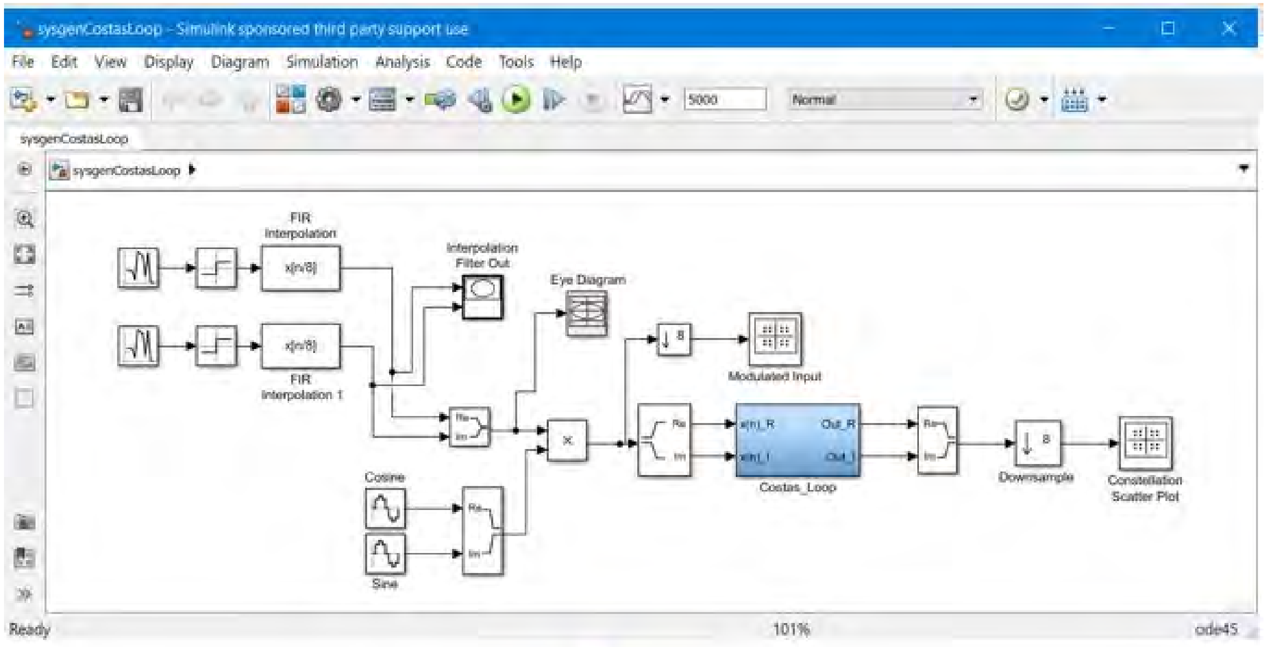


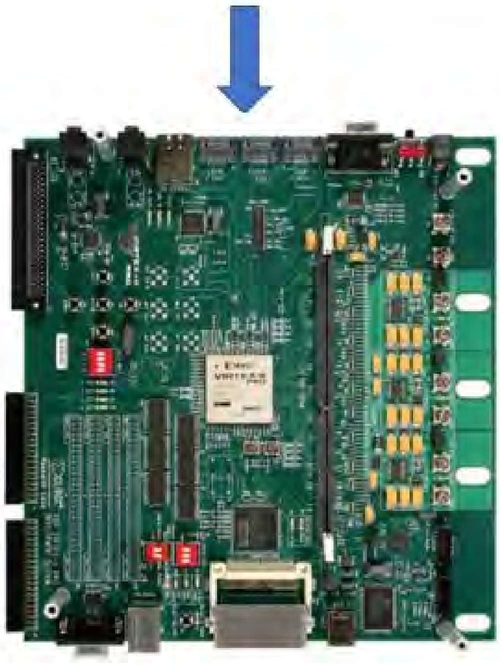
```
function matched = state_machine(din)

persistent state, state = xl_state(0,{xlUnsigned, 3, (

switch state
    case 0
        if din == 1
            state = 1;
        else
            state = 0;
        end
        matched = 0;
```

Hardware Co-Simulation

Model Composer provides accelerated simulation through hardware co-simulation. Model Composer will automatically create a hardware simulation token for a design captured in the Xilinx® HDL blockset that will run on supported hardware platforms. This hardware will co-simulate with the rest of the Simulink® system to provide up to a 1000x simulation performance increase.

Figure 7: **Hardware Co-Simulation**





System Integration Platform

Model Composer provides a system integration platform for the design of DSP FPGAs that allows the RTL, Simulink®, MATLAB® and C/C++ components of a DSP system to come together in a single simulation and implementation environment. Model Composer supports a black box block that allows RTL to be imported into Simulink and co-simulated with either Questa or Xilinx® Vivado® simulator, and provides a Vitis™ HLS block that allows integration and simulation of C/C++ sources.

Post-Installation Tasks

Compiling Xilinx HDL Libraries

The Xilinx tool that compiles libraries for use in Questa SE is named `compile_simlib`.

To compile the Xilinx HDL libraries, launch Vivado and then enter `compile_simlib` in the Vivado Tcl console.

***Note*:** You can enter `compile_simlib -help` in the Vivado Tcl Console for more details on executing this Tcl command.

Managing the Model Composer HDL block Cache

Model Composer incorporates a disk cache to speed up the iterative design process. The cache does this by tagging and storing files related to simulation and generation, then recalling those files during subsequent simulation and generation rather than rerunning the time consuming tools used to create those files.

Specifying Board Support in Model Composer HDL Blockset

When Model Composer is installed on your system as part of a Vivado Design Suite installation, Model Composer will have access to any Xilinx® development boards installed with Vivado.

Additional boards from Xilinx partners are available and a Board Interface file that defines a board (`board.xml`) can be downloaded from a partner website and installed as part of the Vivado Design Suite. You can also create custom Board Interface files, as detailed in Appendix A, Board Interface File, in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895). Both Vivado and Model Composer must be configured to add partner boards and custom boards to the repository of boards available for use.

The procedure for configuring the Vivado Design Suite for use with boards is detailed in Using the Vivado Design Suite Platform Board Flow in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895). The Vivado Design Suite lets you create projects using Xilinx target design platform boards (TDP), or user-specified boards that have been added to a board repository. When you select a specific board, Vivado tools show information about the board, and enable additional designer assistance as part of IP customization, and for IP integrator designs.

To configure Model Composer to use a partner board or custom board, you must add commands to MATLAB®'s `startup.m` file (a file you create for commands to be executed when MATLAB starts up).

To make a board available to your Simulink® models in Model Compsoer:

1. At the MATLAB command line, enter the command `which startup.m` to determine if your MATLAB installation already has a `startup.m` file.

   The `which startup.m` command searches through the folders in the MATLAB search path to find a `startup.m` file. If there is a `startup.m` file in the search path, `which startup.m` displays the full path for the file.

2. Proceed as follows:

   - If your MATLAB installation *does* have a `startup.m` file, enter the command `edit startup.m` at the command line to open the `startup.m` file for editing.

     OR

   - If your MATLAB installation *does not* have a `startup.m` file, create a `startup.m` file in a folder in the MATLAB search path and open the file for editing.

   The command `path` prints a listing of the folders in the search path.

3. Enter the following commands in your `startup.m` file:

   ```
   addpath([getenv('XILINX_VIVADO') '/scripts/sysgen/matlab']);
   xilinx.environment.setBoardFileRepos({'<path1>', '<path2>', '...'}];
   ```

   where the `addpath` command specifies the location of the `setBoardFileRepos` utility and `setBoardFileRepos` points MATLAB to the location of Board Interface files. `<path>` is the path to a folder containing a Board Interface file (`board.xml`) and files referenced by the `board.xml` file, such as `part0_pins.xml` and `preset.xml`. The `<path>` can also specify a folder with multiple subdirectories, each containing a separate Board Interface file.

   For example:

   ```
   addpath([[getenv('XILINX_VIVADO')] '/scripts/sysgen/matlab']);
   xilinx.environment.setBoardFileRepos({'C:/Data/userBoards', 'C:/Data/
   otherBoards'});
   ```

4. Close the `startup.m` file (which is in a directory in the MATLAB search path) and close Model Composer.

When you open Model Composer, each of the partner or custom boards is available as a target board (and target Xilinx device) for your Model Composer design.

To determine what partner or custom boards are available in Model Composer, enter this command in the MATLAB command window:

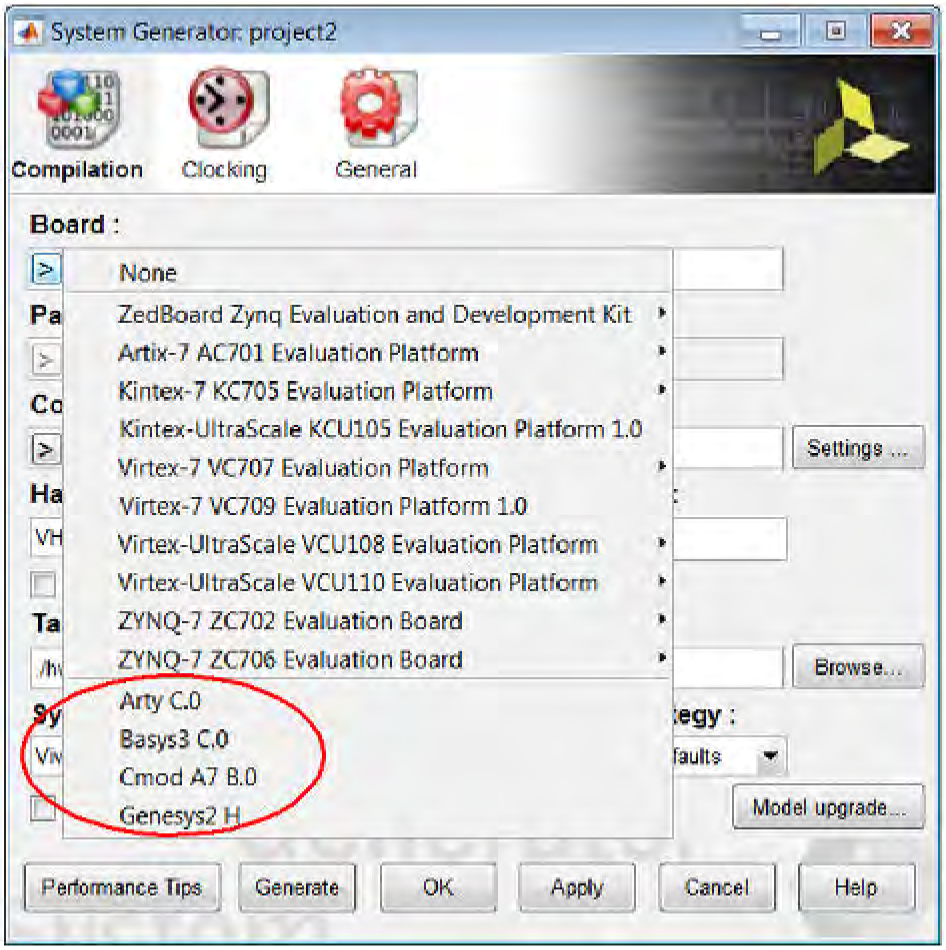
```
xilinx.environment.getBoardFiles
```

A listing of Board Interface files will display in the command window.

```
>> xilinx.environment.getBoardFiles

ans =

    'C:\Data\usrBrds\arty\C.0\board.xml'
    'C:\Data\usrBrds\basys3\C.0\board.xml'
    'C:\Data\usrBrds\cmod_a7\B.0\board.xml'
    'C:\Data\usrBrds\genesys2\H\board.xml'
```

You can also determine what partner or custom boards are available in Model Composer by opening a Simulink® model and double-clicking the model's System Generator token. The added boards will appear in the System Generator token properties dialog box as a **Board** selection:

Figure 8: **Board Selection**


System Generator: project2
Compilation
Clocking
General
Board :
None
ZedBoard Zynq Evaluation and Development Kit
Artix-7 AC701 Evaluation Platform
Kintex-7 KC705 Evaluation Platform
Kintex-UltraScale KCU105 Evaluation Platform 1.0
Virtex-7 VC707 Evaluation Platform
Virtex-7 VC709 Evaluation Platform 1.0
Virtex-UltraScale VCU108 Evaluation Platform
Virtex-UltraScale VCU110 Evaluation Platform
ZYNQ-7 ZC702 Evaluation Board
ZYNQ-7 ZC706 Evaluation Board
Arty C.0
Basys3 C.0
Cmod A7 B.0
Genesys2 H
Settings ...
Browse...
Model upgrade...
Performance Tips
Generate
OK
Apply
Cancel
Help

To add an additional board to your board repository, you can modify the `xilinx.environment.setBoardFileRepos` line in your `startup.m` file to point to the location of the new Board Interface file (`board.xml`). If you place the Board Interface file in a subdirectory under a folder already specified in the `xilinx.environment.setBoardFileRepos` line, the new board will be available the next time you open Model Composer, without having to make any changes to the `startup.m` file.

Hardware Co-Simulation Support

If you have an FPGA development board, you may be able to take advantage of Model Composer's ability to use FPGA hardware co-simulation with Simulink® simulations. The Model Composer software includes support for all Xilinx® Development Boards. Model Composer board support packages can be downloaded from the Boards and Kits page on the Xilinx website.

UNC Paths Not Supported

Model Composer does not support UNC (Universal Naming Convention) paths. For example Model Composer cannot operate on a design that is located on a shared network drive without mapping to the drive first.

Hardware Design Using HDL Library

Model Composer is a system-level modeling tool that facilitates FPGA hardware design. It extends Simulink® in many ways to provide a modeling environment that is well suited to hardware design. The tool provides high-level abstractions that are automatically compiled into an FPGA at the push of a button. The tool also provides access to underlying FPGA resources through low-level abstractions, allowing the construction of highly efficient FPGA designs.

Design Flows Using Model Composer	Describes several settings in which constructing designs in Model Composer is useful.
System-Level Modeling in Model Composer	Discusses Model Composer's ability to implement device-specific hardware designs directly from a flexible, high-level, system modeling environment.
Automatic Code Generation	Discusses automatic code generation for Model Composer designs using the HDL Library.
Compiling MATLAB into an FPGA	Describes how to use a subset of the MATLAB programming language to write functions that describe state machines and arithmetic operators. Functions written in this way can be attached to blocks in Model Composer HDL Library and can be automatically compiled into equivalent HDL.
Importing a Model Composer HDL Design into a Bigger System	Discusses how to take the VHDL netlist from a Model Composer design and synthesize it in order to embed it into a larger design. Also shows how VHDL created by Model Composer can be incorporated into a simulation model of the overall system.
Configurable Subsystems and Model Composer	Explains how to use configurable Subsystems in Model Composer. Describes common tasks such as defining configurable Subsystems, deleting and adding blocks, and using configurable Subsystems to import compilation results into Model Composer designs.
Notes for Higher Performance FPGA Design	Suggests design practices in Model Composer that lead to an efficient and high-performance implementation in an FPGA.
Using the FDATool in Digital Filter Applications	Demonstrates one way to specify, implement and simulate a FIR filter using the FDATool block.
Multiple Independent Clocks Hardware Design	The design can be partitioned into groups of Subsystem blocks, where each Subsystem has a common cycle period, independent of the cycle period of other Subsystems.
AXI Interface	Provides an introduction to AMBA AXI4 and draws attention to AMBA AXI4 details with respect to Model Composer
AXI4-Lite Slave Interface Generation	Describes features in Model Composer that allow you to create a standard AXI4-Lite interface for a Model Composer module and then export the module to the Vivado® IP catalog for later inclusion in a larger design using IP integrator.

Tailor Fitting a Platform Based Accelerator Design in Model Composer	Describes how to develop an accelerator in Model Composer which is part of a platform framework developed in the Vivado IP Integrator.

Design Flows Using Model Composer

Model Composer can be useful in many settings. Sometimes you may want to explore an algorithm without translating the design into hardware. Other times you might plan to use a Model Composer design as part of something bigger. A third possibility is that a Model Composer design is complete in its own right, and is to be used in FPGA hardware. This topic describes all three possibilities.

Algorithm Exploration

Model Composer is particularly useful for algorithm exploration, design prototyping, and model analysis. When these are the goals, you can use the tool to flesh out an algorithm in order to get a feel for the design problems that are likely to be faced, and perhaps to estimate the cost and performance of an implementation in hardware. The work is preparatory, and there is little need to translate the design into hardware.

In this setting, you assemble key portions of the design without worrying about fine points or detailed implementation. Simulink blocks and MATLAB M-code provide stimuli for simulations, and for analyzing results. Resource estimation gives a rough idea of the cost of the design in hardware. Experiments using hardware generation can suggest the hardware speeds that are possible.

Once a promising approach has been identified, the design can be fleshed out. Model Composer allows refinements to be done in steps, so some portions of the design can be made ready for implementation in hardware, while others remain high-level and abstract. Model Composer's facilities for hardware co-simulation are particularly useful when portions of a design are being refined.

Implementing Part of a Larger Design

Often Model Composer is used to implement a portion of a larger design. For example, Model Composer is a good setting in which to implement data paths and control, but is less well suited for sophisticated external interfaces that have strict timing requirements. In this case, it may be useful to implement parts of the design using Model Composer, implement other parts outside, and then combine the parts into a working whole.

A typical approach to this flow is to create an HDL wrapper that represents the entire design, and to use the Model Composer portion as a component. The non-Model Composer portions of the design can also be components in the wrapper, or can be instantiated directly in the wrapper.

Implementing a Complete Design

Many times, everything needed for a design is available inside Model Composer. For such a design, pressing the **Generate** button instructs Model Composer to translate the design into HDL, and to write the files needed to process the HDL using downstream tools. The files written include the following:

- HDL that implements the design itself.
- An HDL test bench. The test bench allows results from Simulink simulations to be compared against ones produced by a logic simulator.
- Files that allow the Model Composer HDL to be used as a Vivado IDE project.

For details concerning the files that Model Composer writes, see the topic Compilation Results.

Note to DSP Engineers

Model Composer extends Simulink to enable hardware design, providing high-level abstractions that can be automatically compiled into an FPGA. Although the arithmetic abstractions are suitable to Simulink (discrete time and space dynamical system simulation), Model Composer also provides access to features in the underlying FPGA.

The more you know about a hardware realization (e.g., how to exploit parallelism and pipelining), the better the implementation you'll obtain. Using IP cores makes it possible to have efficient FPGA designs that include complex functions like FFTs. Model Composer also makes it possible to refine a model to more accurately fit the application.

Scattered throughout the Model Composer documentation are notes that explain ways in which system parameters can be used to exploit hardware capabilities.

Note to Hardware Engineers

Model Composer does not replace hardware description language (HDL)-based design, but does makes it possible to focus your attention only on the critical parts. By analogy, most DSP programmers do not program exclusively in assembler; they start in a higher-level language like C, and write assembly code only where it is required to meet performance requirements.

A good rule of thumb is this: in the parts of the design where you must manage internal hardware clocks (e.g., using DDR or phased clocking), you should implement using HDL. The less critical portions of the design can be implemented in Model Composer, and then the HDL and Model Composer portions can be connected. Usually, most portions of a signal processing system do not need this level of control, except at external interfaces. Model Composer provides mechanisms to import HDL code into a design (see Importing HDL Modules) that are of particular interest to the HDL designer.

Another aspect of Model Composer that is of interest to engineers who design using HDL is its ability to automatically generate an HDL test bench, including test vectors. This aspect is described in the topic HDL Testbench.

Finally, the hardware co-simulation interfaces described in the topic Using Hardware Co-Simulation allow you to run a design in hardware under the control of Simulink, bringing the full power of MATLAB and Simulink to bear for data analysis and visualization.

System-Level Modeling in Model Composer

Model Composer allows device-specific hardware designs to be constructed directly in a flexible high-level system modeling environment. In a Model Composer design, signals are not just bits. They can be signed and unsigned fixed-point numbers, and changes to the design automatically translate into appropriate changes in signal types. Blocks are not just stand-ins for hardware. They respond to their surroundings, automatically adjusting the results they produce and the hardware they become.

Model Composer allows designs to be composed from a variety of ingredients. Data flow models, traditional hardware description languages (VHDL and Verilog), and functions derived from the MATLAB programming language, can be used side-by-side, simulated together, and synthesized into working hardware. Model Composer HDL block simulation results are bit and cycle-accurate. This means results seen in simulation exactly match the results that are seen in hardware. Model Composer simulations are considerably faster than those from traditional HDL simulators, and results are easier to analyze.

Model Composer HDL Blocksets	Describes how Model Composer's HDL blocks are organized in libraries, and how the blocks can be parameterized and used.
Xilinx Commands that Facilitate Rapid Model Creation and Analysis	Introduces Xilinx commands that have been added to the Simulink pop-up menu that facilitate rapid Model Composer model creation and analysis.
Signal Types	Describes the data types used by Model Composer and ways in which data types can be automatically assigned by the tool.
Bit-True and Cycle-True Modeling	Specifies the relationship between the Simulink-based simulation of a Model Composer model and the behavior of the hardware that can be generated from it.
Timing and Clocking	Describes how clocks are implemented in hardware, and how their implementation is controlled inside Model Composer. Explains how Model Composer translates a multirate Simulink model into working clock-synchronous hardware.
Synchronization Mechanisms	Describes mechanisms that can be used to synchronize data flow across the data path elements in a high-level Model Composer design, and describes how control path functions can be implemented.
Block Masks and Parameter Passing	Explains how parameterized systems and Subsystems are created in Simulink.

Model Composer HDL Blocksets

A Simulink® blockset is a library of blocks that can be connected in the Simulink block editor to create functional models of a dynamical system. For system modeling, Model Composer HDL library blocksets are used like other Simulink blocksets. The blocks provide abstractions of mathematical, logical, memory, and DSP functions that can be used to build sophisticated signal processing (and other) systems. There are also blocks that provide interfaces to other software tools (e.g. FDATool, Questa) as well as the Model Composer code generation software.

Model Composer HDL blocks are *bit-accurate* and *cycle-accurate*. Bit-accurate blocks produce values in Simulink that match corresponding values produced in hardware; cycle-accurate blocks produce corresponding values at corresponding times.

Xilinx HDL Blockset

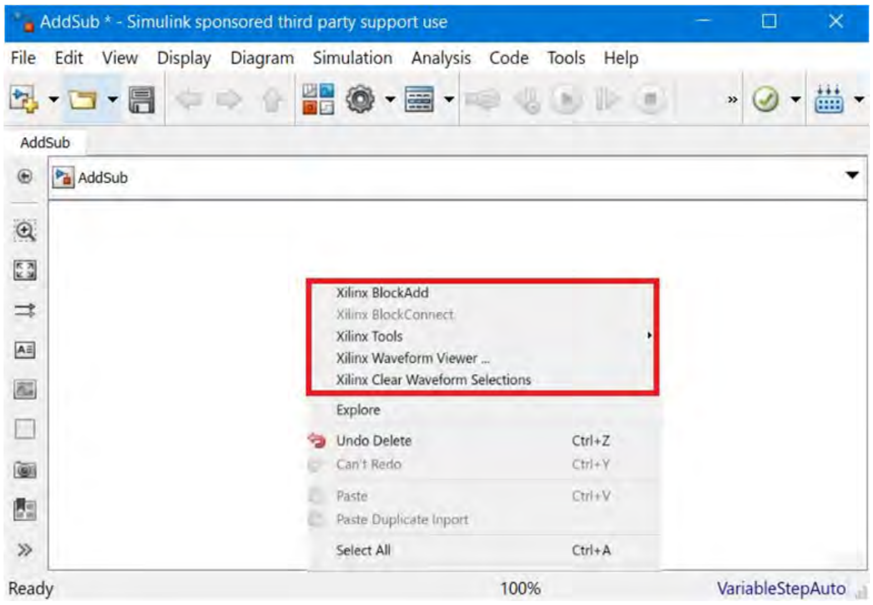
The Xilinx® HDL Blockset is a family of libraries that contain basic Model Composer HDL blocks. Some blocks are low-level, providing access to device-specific hardware. Others are high- level, implementing (for example) signal processing and advanced communications algorithms. The libraries are described in the following table.

Library	Description
Basic Elements	Standard building blocks for digital logic.
DSP	Digital signal processing (DSP) blocks.
Interfaces	Blocks that support connection with Simulink blocks.
Logic and Bit Operations	Blocks for performing logic and bit operations.
Memory	Blocks that implement and access memories.
Signal Routing	Blocks that support Routing signals.
Sources	Blocks that generate signal data
Tools	“Utility” blocks. For example, code generation (System Generator token), resource estimation, HDL co-simulation, etc.
User-Defined functions	Blocks that support importing custom functions.

Xilinx Commands that Facilitate Rapid Model Creation and Analysis

Xilinx has added graphics commands to the Simulink® popup menu that will help you rapidly create and analyze your Model Composer design. As shown below, you can access these commands by right-clicking the Simulink model canvas and selecting the appropriate Xilinx command:

Figure 9: **Xilinx Commands**


AddSub * - Simulink sponsored third party support use
File Edit View Display Diagram Simulation Analysis Code Tools Help
AddSub
AddSub
Xilinx BlockAdd
Xilinx BlockConnect
Xilinx Tools
Xilinx Waveform Viewer ...
Xilinx Clear Waveform Selections
Explore
Undo Delete Ctrl+Z
Can't Redo Ctrl+Y
Paste Ctrl+V
Paste Duplicate Inport
Select All Ctrl+A
Ready
100%
VariableStepAuto

Details on how to use these additional Xilinx commands are provided in the topics for each individual command.

Signal Types

In order to provide bit-accurate simulation of hardware, HDL blocks operate on Boolean, floating-point, and arbitrary precision fixed-point values. By contrast, the fundamental scalar signal type in Simulink® is double precision floating point. The gateway blocks in the Model Composer HDL library allow connection between HDL blocks in the Xilinx toolbox and blocks in the Simulink library . The Gateway In converts a double precision signal into a Xilinx signal, and the Gateway Out converts a Xilinx signal into double precision. Simulink® continuous time signals must be sampled by the Gateway In block.

Most HDL blocks are polymorphic, i.e., they can deduce appropriate output types based on their input types. When *full precision* is specified for a block in its parameters dialog box, Model Composer chooses the output type to ensure no precision is lost. Sign extension and zero padding occur automatically as necessary. *User-specified precision* is usually also available. This allows you to set the output type for a block and to specify how quantization and overflow should be handled. Quantization possibilities include unbiased rounding towards plus or minus infinity, depending on sign, or truncation. Overflow options include saturation, truncation, and reporting overflow as an error.

Note: Model Composer data types can be displayed by selecting **Display → Signals & Ports → Port Data Types** in Simulink. Displaying data types makes it easy to determine precision throughout a model. If, for example, the type for a port is **Fix_11_9**, then the signal is a two's complement signed 11-bit number having nine fractional bits. Similarly, if the type is **Ufix_5_3**, then the signal is an unsigned 5-bit number having three fractional bits.

In the Model Composer portion of a Simulink model, every signal must be sampled. Sample times may be inherited using Simulink's propagation rules, or set explicitly in a block customization dialog box. When there are feedback loops, Model Composer is sometimes unable to deduce sample periods and/or signal types, in which case the tool issues an error message. Assert blocks must be inserted into loops to address this problem. It is not necessary to add assert blocks at every point in a loop; usually it suffices to add an assert block at one point to "break" the loop.

***Note*:** Simulink can display a model by shading blocks and signals that run at different rates with different colors (**Display → Sample Time → Colors** in the Simulink pulldown menus). This is often useful in understanding multirate designs.

Floating-Point Data Type

Many Model Composer HDL blocks across various libraries support the floating-point data type.

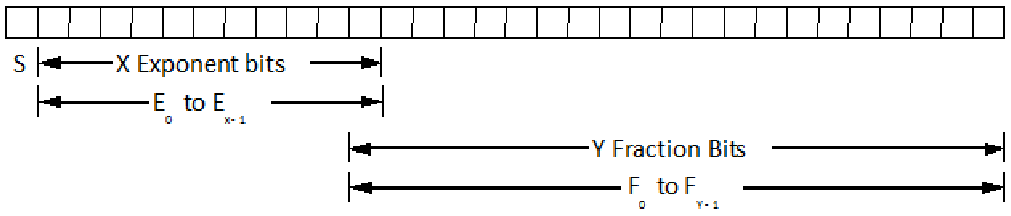
Model Composer uses the Floating-Point Operator v7.1 IP core to leverage the implementation of operations such as addition/subtraction, multiplication, comparisons and data type conversion.

The floating-point data type support is in compliance with IEEE-754 Standard for Floating-Point Arithmetic. Single precision, Double precision and Custom precision floating-point data types are supported for design input, data type display and for data rate and type propagation (RTP) across the supported HDL blocks.

IEEE-754 Standard for Floating-Point Data Type

As shown below, floating-point data is represented using one Sign bit (S), X exponent bits and Y fraction bits. The Sign bit is always the most-significant bit (MSB).

Figure 10: **Floating-Point Data**



According to the IEEE-754 standard, a floating-point value is represented and stored in the normalized form. In the normalized form the exponent value E is a biased/normalized value. The normalized exponent, E, equals the sum of the actual exponent value and the exponent bias. In the normalized form, Y-1 bits are used to store the fraction value. The F0 fraction bit is always a hidden bit and its value is assumed to be 1.

S represents the value of the sign of the number. If S is 0 then the value is a positive floating-point number; otherwise it is negative. The X bits that follow are used to store the normalized exponent value E and the last Y-1 bits are used to store the fraction/mantissa value in the normalized form.

For the given exponent width, the exponent bias is calculated using the following equation:

$$\text{Exponent\_bias} = 2^{(X - 1)} - 1$$

Where X is the exponent bit width.

According to the IEEE standard, a single precision floating-point data is represented using 32 bits. The normalized exponent and fraction/mantissa are allocated 8 and 24 bits, respectively. The exponent bias for single precision is 127. Similarly, a double precision floating-point data is represented using a total of 64 bits where the exponent bit width is 11 and the fraction bit width is 53. The exponent bias value for double precision is 1023.

The normalized floating-point number in the equation form is represented as follows:

$$\text{Normalized Floating-Point Value} = (-1)^{S} \text{ x F0.F1F2 . FY-2FY-1 x } (2)^{E}$$

The actual value of exponent (E_actual) = E - Exponent_bias. Considering 1 as the value for the hidden bit F0 and the E_actual value, a floating-point number can be calculated as follows:

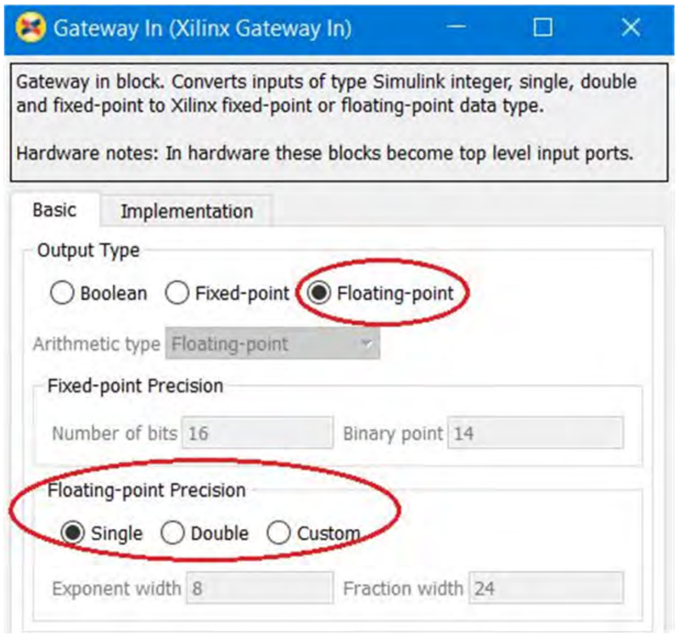
$$\text{FP\_Value} = (-1)^{S} \text{ x 1.F1F2 . FY-2FY-1 x } (2)^{(\text{E\_actual})}$$

Floating-Point Data Representation in Model Composer

The HDL Gateway In block supports Boolean, Fixed-point, and Floating-point data types as shown in the following figure. You can select either a **Single**, **Double** or **Custom** precision type after specifying the floating-point data type.

For example, if Exponent width of 9 and Fraction width of 31 is specified then the floating-point data value will be stored in total 40 bits where the MSB bit will be used for sign representation, the following 9 bits will be used to store biased exponent value and the 30 LSB bits will be used to store the fractional value.

Figure 11: **Floating-point Precision**


Gateway In (Xilinx Gateway In)

Gateway in block. Converts inputs of type Simulink integer, single, double and fixed-point to Xilinx fixed-point or floating-point data type.

Hardware notes: In hardware these blocks become top level input ports.

Output Type: Boolean Fixed-point [✓ Floating-point]

Arithmetic type: Floating-point

Fixed-point Precision

Number of bits: 16 Binary point: 14

Floating-point Precision: [✓ Single] Double Custom

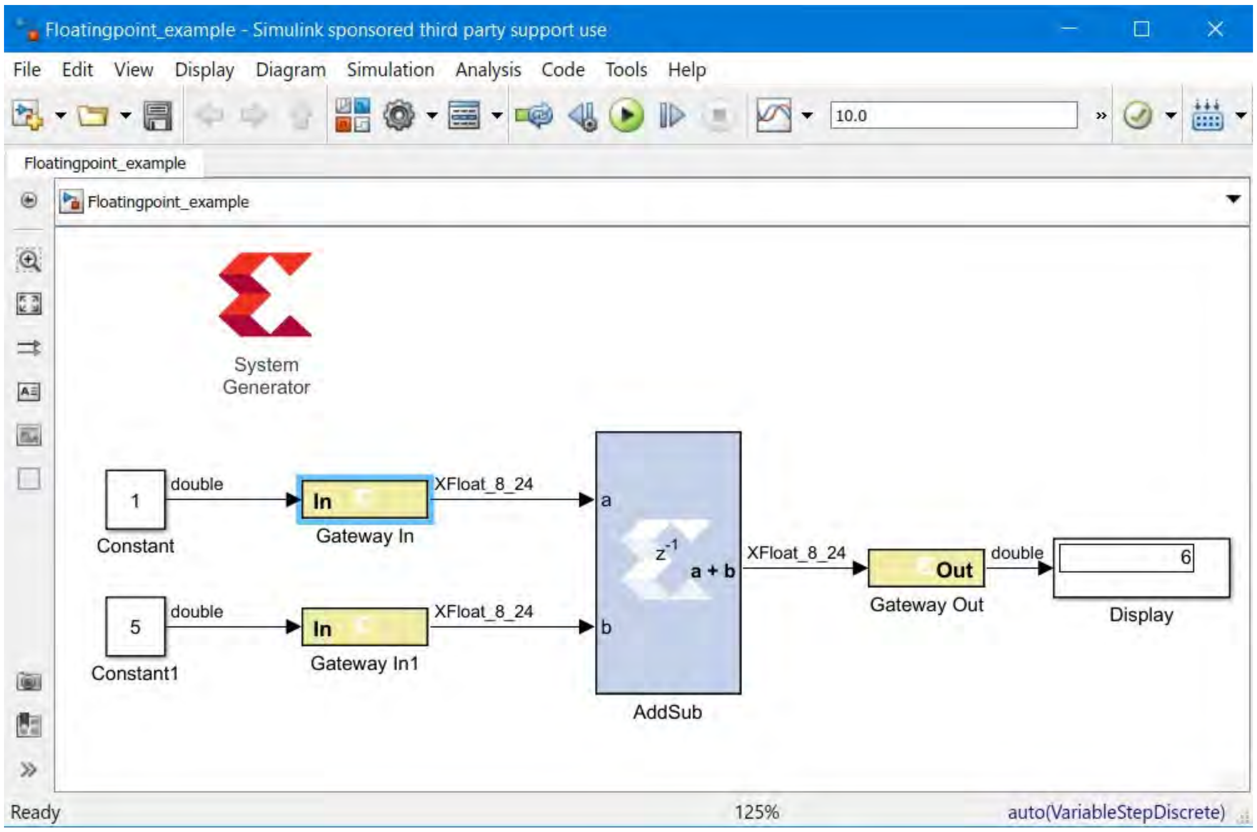
Exponent width: 8 Fraction width: 24

In compliance with the IEEE-754 standard, if **Single** precision is selected then the total bit width is assumed to be 32; 8 bits for the exponent and 24 bits for the fraction. Similarly when **Double** precision is selected, the total bit width is assumed to be 64 bits; 11 bits for the exponent and 53 bits for the fraction part. When **Custom** precision is selected, the **Exponent width** and **Fraction width** fields are activated and you are free to specify values for these fields (8 and 24 are the default values). The total bit width for **Custom** precision data is the summation of the number of exponent bits and the number of fraction bits. Similar to fraction bit width for **Single** precision and **Double** precision data types the fraction bit width for **Custom** precision data type must include the hidden bit F0.

Displaying the Data Type on Output Signals

As shown below, after a successful rate and type propagation, the floating-point data type is displayed on the output of each HDL block. To display the signal data type as shown in the diagram below, you select the pulldown menu item **Display→Signals & Ports→Port Data Types**.

Figure 12: **Floating-point Data Type**



A floating-point data type is displayed using the format: `XFloat_<exponent_bit_width>_<fraction_bit_width>`. Single and Double precision data types are displayed using the string "`XFloat_8_24`" and "`XFloat_11_53`", respectively.

If for a Custom precision data type the exponent bit width 9 and the fraction bit width 31 are specified, then it will be displayed as "`XFloat_9_31`". A total of 40 bits will be used to store the floating-point data value. Because floating-point data is stored in a normalized form, the fractional value will be stored in 30 bits.

In Model Composer the fixed-point data type is displayed using format `XFix_<total_data_width>_<binary_point_width>`. For example, a fixed-point data type with the data width of 40 and binary point width of 31 is displayed as `XFix_40_31`.

It is necessary to point out that in the fixed-point data type the actual number of bits used to store the fractional value is different from that used for floating-point data type. In the example above, all 31 bits are used to store the fractional bits of the fixed-point data type.

Model Composer uses the exponent bit width and the fraction bit width to configure and generate an instance of the Floating-Point Operator core.

Rate and Type Propagation

During data rate and type propagation across a Model Composer HDL block that supports floating-point data, the following design rules are verified. The appropriate error is issued if one of the following violations is detected.

1. If a signal carrying floating-point data is connected to the port of an HDL block that doesn't support the floating-point data type.
2. If the data input (both A and B data inputs, where applicable) and the data output of an HDL block are not of the same floating-point data type. The DRC check will be made between the two inputs of a block as well as between an input and an output of the block.

   If a Custom precision floating-point data type is specified, the exponent bit width and the fraction bit width of the two ports are compared to determine that they are of the same data type.

   ***Note*:** The Convert and Relational blocks are excluded from this check. The Convert block supports Float-to-float data type conversion between two different floating-point data types. The Relational block output is always the Boolean data type because it gives a true or false result for a comparison operation.
3. If the data inputs are of the fixed-point data type and the data output is expected to be floating-point and vice versa.

   ***Note*:** The Convert and Relational blocks are excluded from this check. The Convert block supports Fixed-to-float as well as Float-to-fixed data type conversion. The Relational block output is always the Boolean data type because it gives a true or false result for a comparison operation.
4. If Custom precision is selected for the Output Type of blocks that support the floating-point data type. For example, for blocks such as AddSub, Mult, CMult, and MUX, only Full output precision is supported if the data inputs are of the floating-point data type.
5. If the Carry In port or Carry Out port is used for the AddSub block when the operation on a floating-point data type is specified.
6. If the Floating-Point Operator IP core gives an error for DRC rules defined for the IP.

AXI Signal Groups

Model Composer HDL blocks found in the DSP library contain interfaces that conform to the AXI4 specification. Blocks with AXI4 interfaces are drawn such that ports relating to a particular AXI4 interface are grouped and colored in similarly. This makes it easier to identify data and control signals pertaining to the same interface. Grouping similar AXI4 ports together also make it possible to use the Simulink Bus Creator and Simulink Bus Selector blocks to connect groups of signals together. More information on AXI4 can be found in the section titled AXI Interface. For more detailed information on the AMBA AXI4 specification, please refer to the Xilinx AMBA AXI4 documents found at the AMBA AXI4 Interface Protocol page on the Xilinx website.

Bit-True and Cycle-True Modeling

Simulations in Model Composer are *bit-true* and *cycle-true*. To say a simulation is bit-true means that at the boundaries (i.e., interfaces between Model Composer HDL blocks and non-HDL blocks), a value produced in simulation is bit-for-bit identical to the corresponding value produced in hardware. To say a simulation is cycle-true means that at the boundaries, corresponding values are produced at corresponding times. The boundaries of the design are the points at which Model Composer HDL gateway blocks exist. When a design is translated into hardware, Gateway In (respectively, Gateway Out) blocks become top-level input (resp., output) ports.

Timing and Clocking

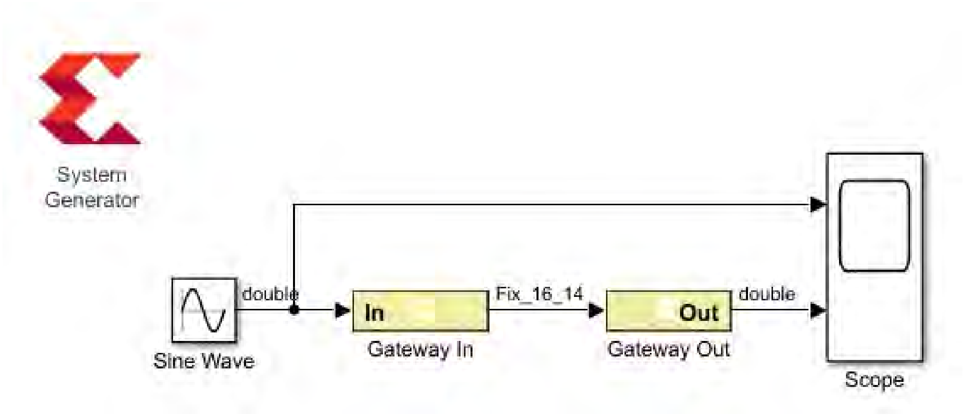
Discrete Time Systems

Designs in Model Composer are discrete time systems. In other words, the signals and the blocks that produce them have associated sample rates. A block's sample rate determines how often the block is awoken (allowing its state to be updated). Model Composer sets most sample rates automatically. A few blocks, however, set sample rates explicitly or implicitly.

***Note*:** For an in-depth explanation of Simulink discrete time systems and sample times, consult the Using Simulink reference manual from the MathWorks, Inc.

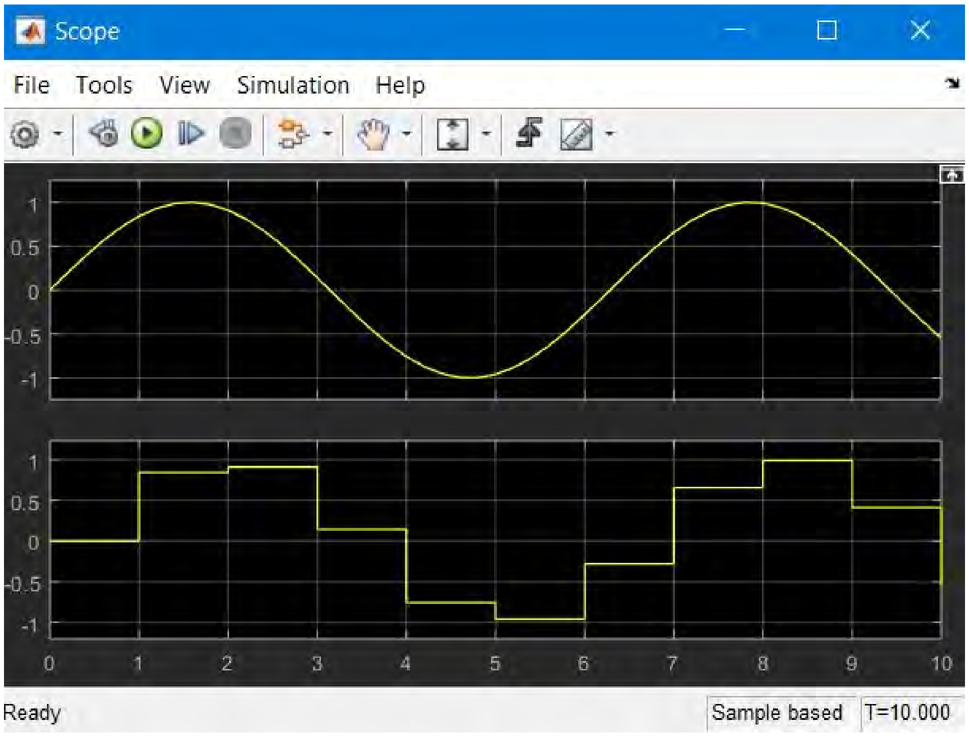
A simple Model Composer model illustrates the behavior of discrete time systems. Consider the model shown below. It contains a gateway that is driven by a Simulink source (Sine Wave), and a second gateway that drives a Simulink sink (Scope).

Figure 13: **Discrete Time System**



The Gateway In block is configured with a sample period of one second. The Gateway Out block converts the Xilinx fixed-point signal back to a double (so it can analyzed in the Simulink scope), but does not alter sample rates. The scope output below shows the unaltered and sampled versions of the sine wave.

Figure 14: **Scope Output**



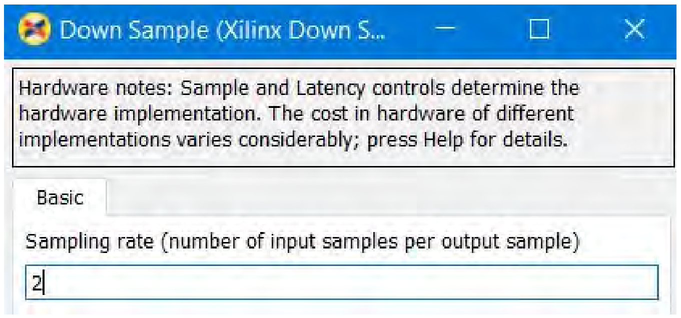
Multirate Models

Model Composer supports *multirate* designs, i.e., designs having signals running at several sample rates. Model Composer automatically compiles multirate models into hardware. This allows multirate designs to be implemented in a way that is both natural and straightforward in Simulink.

Rate-Changing Blocks

The Model Composer HDL library includes blocks that change sample rates. The most basic rate changers are the Up Sample and Down Sample blocks. As shown in the following figure, these blocks explicitly change the rate of a signal by a fixed multiple that is specified in the block's dialog box.

Figure 15: **Rate Change Dialog**



Hardware notes: Sample and Latency controls determine the hardware implementation. The cost in hardware of different implementations varies considerably; press Help for details.

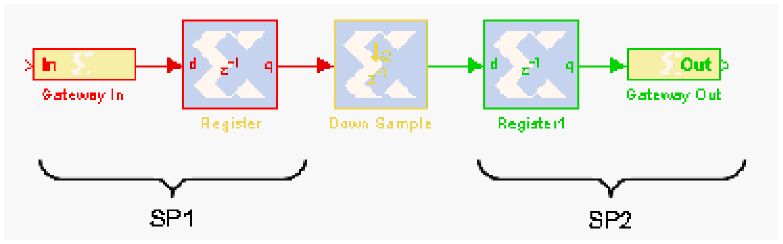
Sampling rate (number of input samples per output sample)

2

Other blocks (e.g., the Parallel To Serial and Serial To Parallel converters) change rates implicitly in a way determined by block parameterization.

Consider the simple multirate example below. This model has two sample periods, SP1 and SP2. The Gateway In dialog box defines the sample period SP1. The Down Sample block causes a rate change in the model, creating a new rate SP2 which is half as fast as SP1.

Figure 16: **Multirate Example**



Hardware Oversampling

Some HDL blocks are oversampled, i.e., their internal processing is done at a rate that is faster than their data rates. In hardware, this means that the block requires more than one clock cycle to process a data sample. In Simulink such blocks do not have an observable effect on sample rates.

Although blocks that are oversampled do not cause an explicit sample rate change in Simulink, Model Composer considers the internal block rate along with all other sample rates when generating clocking logic for the hardware implementation. This means that you must consider the internal processing rates of oversampled blocks when you specify the **Simulink system period** value in the System Generator token dialog box.

Asynchronous Clocking

Model Composer focuses on the design of hardware that is synchronous to a single clock. It can, under some circumstances, be used to design systems that contain more than one clock. This is possible provided the design can be partitioned into individual clock domains with the exchange of information between domains being regulated by dual port memories and FIFOs. The remainder of this topic focuses exclusively on the clock-synchronous aspects of Model Composer. This discussion is relevant to both single-clock and multiple-clock designs.

Synchronous Clocking

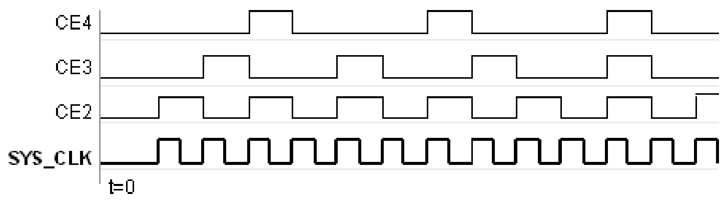
By default, Model Composer creates designs with synchronous clocking, where multiple rates are realized using clock enables. When Model Composer compiles a model into hardware, it preserves the sample rate information of the design in such a way that corresponding portions in hardware run at appropriate rates. In hardware, Model Composer generates related rates by using a single clock in conjunction with clock enables, one enable per rate. The period of each clock enable is an integer multiple of the period of the system clock.

Inside Simulink, neither clocks nor clock enables are required as explicit signals in a Model Composer design. When Model Composer compiles a design into hardware, it uses the sample rates in the design to deduce what clock enables are needed. To do this, it employs two user-specified values from the System Generator token: the **Simulink system period** and **FPGA clock period**. These numbers define the scaling factor between time in a Simulink simulation, and time in the actual hardware implementation. The Simulink system period must be the greatest common divisor (gcd) of the sample periods that appear in the model, and the FPGA clock period is the period, in nanoseconds, of the system clock. If p represents the Simulink system period, and c represents the FPGA system clock period, then something that takes kp units of time in Simulink takes k ticks of the system clock (hence kc nanoseconds) in hardware.

To illustrate this point, consider a model that has three Simulink sample periods 2, 3, and 4. The gcd of these sample periods is 1, and should be specified as such in the **Simulink system period** field for the model. Assume the **FPGA clock period** is specified to be 10ns. With this information, the corresponding clock enable periods can be determined in hardware.

In hardware, we refer to the clock enables corresponding to the Simulink sample periods 2, 3, and 4 as CE2, CE3, and CE4, respectively. The relationship of each clock enable period to the system clock period can be determined by dividing the corresponding Simulink sample period by the Simulink System Period value. Thus, the periods for CE2, CE3, and CE4 equal 2, 3, and 4 system clock periods, respectively. A timing diagram for the example clock enable signals is shown below:

Figure 17: **Timing Diagram**



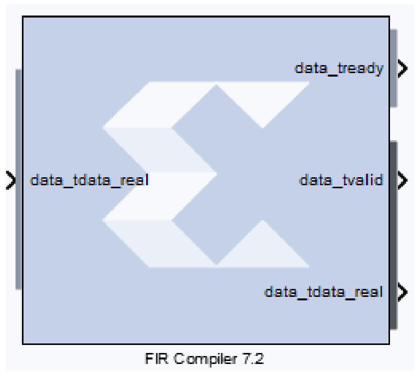
Synchronization Mechanisms

Model Composer does not make implicit synchronization mechanisms available. Instead, synchronization is the responsibility of the designer, and must be done explicitly.

Valid Ports

Model Composer provides several blocks (in particular, the AXI FIFO) that can be used for synchronization. Several blocks provide optional AXI signaling interfaces to denote when a sample is valid (TValid) and when the interface is ready for data (TReady). Note that the tvalid / tready ports may not be visible based on the configuration of the IP. Color association denotes a collection of ports for each interface on the block as shown below. Blocks with interfaces can be chained, affording a primitive form of flow control. Examples of such blocks with AXI interfaces include the FFT, FIR, and DDS.

Figure 18: **Block with AXI Interface**



Indeterminate Data

Indeterminate values are common in many hardware simulation environments. Often they are called "don't cares" or "Xs". In particular, values in Model Composer simulations can be indeterminate. A dual port memory block, for example, can produce indeterminate results if both ports of the memory attempt to write the same address simultaneously. What actually happens in hardware depends upon effectively random implementation details that determine which port sees the clock edge first. Allowing values to become indeterminate gives the system designer greater flexibility. Continuing the example, there is nothing wrong with writing to memory in an indeterminate fashion if subsequent processing does not rely on the indeterminate result.

HDL modules that are brought into the simulation through HDL co-simulation are a common source for indeterminate data samples. Model Composer presents indeterminate values to the inputs of an HDL co-simulating module as the standard logic vector 'XXX . . . XX'.

Indeterminate values that drive a Gateway Out become what are called NaNs. (NaN abbreviates "not a number".) In a Simulink scope, NaN values are not plotted. Conversely, NaNs that drive a Gateway In become indeterminate values. Model Composer provides an Indeterminate Probe block that allows for the detection of indeterminate values. This probe cannot be translated into hardware.

In Model Composer, any arithmetic signal can be indeterminate, but Boolean signals cannot be. If a simulation reaches a condition that would force a Boolean to become indeterminate, the simulation is halted and an error is reported. Many Xilinx blocks have control ports that only allow Boolean signals as inputs. The rule concerning indeterminate Booleans means that such blocks never see an indeterminate on a control port

A UFix_1_0 is a type that is equivalent to Boolean except for the above restriction concerning indeterminate data.

Block Masks and Parameter Passing

The same scoping and parameter passing rules that apply to ordinary Simulink blocks apply to HDL blocks. Consequently, blocks in the Xilinx HDL Blockset can be parameterized using MATLAB variables and expressions. This capability makes possible highly parametric designs that take advantage of the expressive and computational power of the MATLAB language.

Block Masks

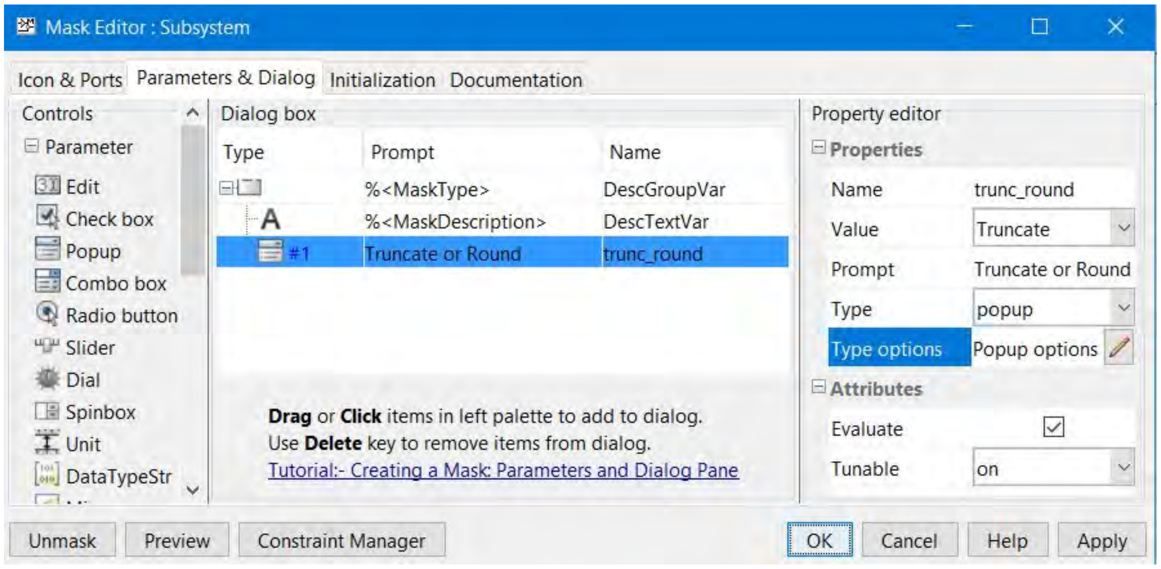
In Simulink, blocks are parameterized through a mechanism called *masking*. In essence, a block can be assigned *mask variables* whose values can be specified by a user through dialog box prompts or can be calculated in mask initialization commands. Variables are stored in a *mask workspace*. A mask workspace is local to the blocks under the mask and cannot be accessed by external blocks.

Note: It is possible for a mask to access global variables and variables in the base workspace. To access a base workspace variable, use the MATLAB evalin function. For more information on the MATLAB and Simulink scoping rules, refer to the manuals titled *Using MATLAB* and *Using Simulink* from The MathWorks, Inc.

Parameter Passing

It is often desirable to pass variables to blocks inside a masked Subsystem. Doing so allows the block's configuration to be determined by parameters on the enclosing Subsystem. This technique can be applied to parameters on blocks in the Xilinx HDL blockset whose values are set using a listbox, radio button, or checkbox. For example, when building a Subsystem that consists of a multiply and accumulate block, you can create a parameter on the Subsystem that allows you to specify whether to truncate or round the result. This parameter will be called trunc_round as shown in the figure below.

Figure 19: **Creating a Parameter**



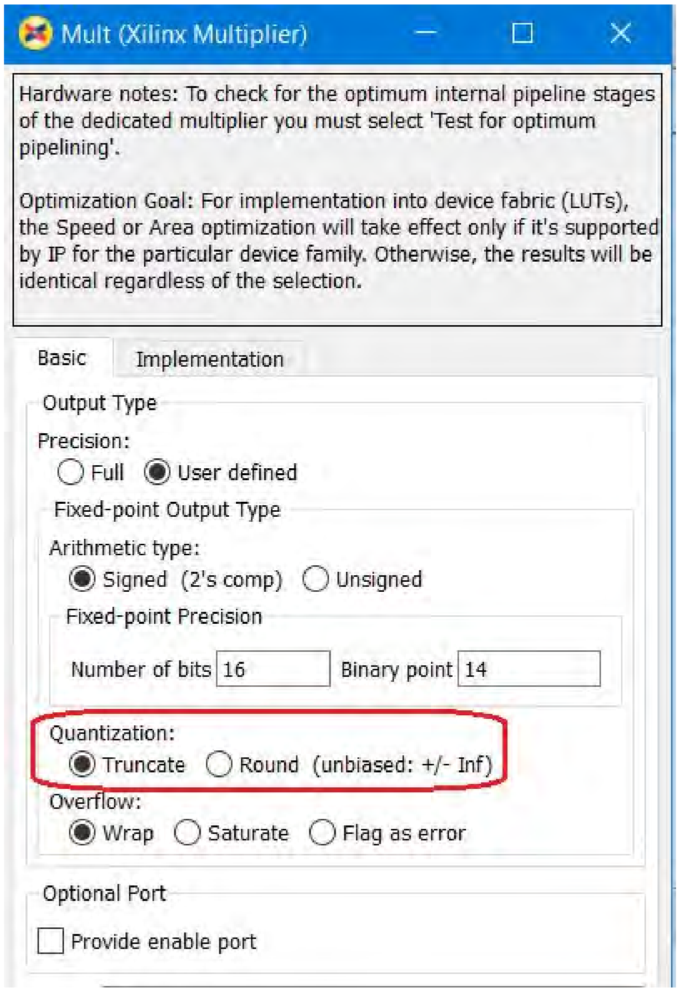
Type	Prompt	Name
	%<MaskType>	DescGroupVar
A	%<MaskDescription>	DescTextVar
#1	Truncate or Round	trunc_round

Drag or **Click** items in left palette to add to dialog.
Use **Delete** key to remove items from dialog.
Tutorial:- Creating a Mask: Parameters and Dialog Pane

Name	trunc_round
Value	Truncate
Prompt	Truncate or Round
Type	popup
Type options	Popup options
Evaluate	✓
Tunable	on

As shown below, in the parameter editing dialog for the accumulator and multiplier blocks, there are radio buttons that allow either the truncate or round option to be selected.

Figure 20: **Editing a Parameter**


Mult (Xilinx Multiplier)

Hardware notes: To check for the optimum internal pipeline stages of the dedicated multiplier you must select 'Test for optimum pipelining'.

Optimization Goal: For implementation into device fabric (LUTs), the Speed or Area optimization will take effect only if it's supported by IP for the particular device family. Otherwise, the results will be identical regardless of the selection.

Basic | Implementation

Output Type

Precision: Full [✓ User defined]

Fixed-point Output Type

Arithmetic type: [✓ Signed (2's comp)] Unsigned

Fixed-point Precision

Number of bits 16 Binary point 14

Quantization: [✓ Truncate] Round (unbiased: +/- Inf)

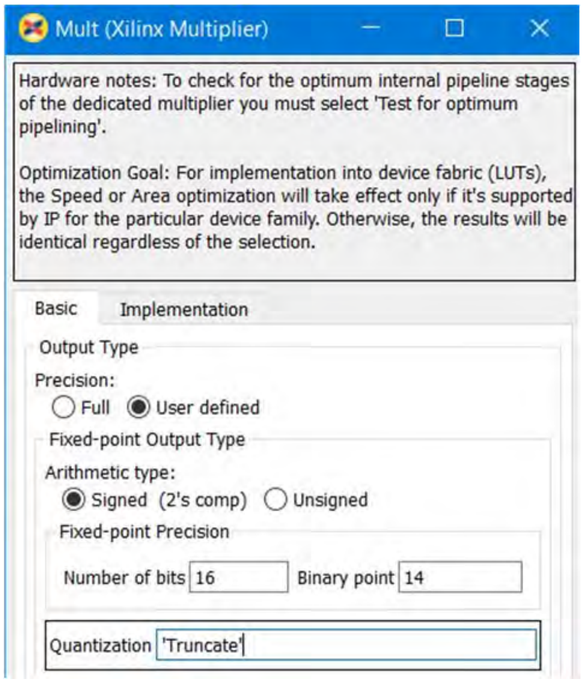
Overflow: [✓ Wrap] Saturate Flag as error

Optional Port

Provide enable port

In order to use a parameter rather than the radio button selection, right-click the radio button and select **Define With Expression**. A MATLAB expression can then be used as the parameter setting. In the example below, the trunc_round parameter from the Subsystem mask can be used in both the accumulator and multiply blocks so that each block will use the same setting from the mask variable on the Subsystem.

Figure 21: **Using a Parameter**



Hardware notes: To check for the optimum internal pipeline stages of the dedicated multiplier you must select 'Test for optimum pipelining'.

Optimization Goal: For implementation into device fabric (LUTs), the Speed or Area optimization will take effect only if it's supported by IP for the particular device family. Otherwise, the results will be identical regardless of the selection.

Precision: ( ) Full [✓ User defined]

Arithmetic type: [✓ Signed (2's comp)] ( ) Unsigned

Number of bits 16 Binary point 14

Quantization 'Truncate'

Automatic Code Generation

Model Composer automatically compiles designs into low-level representations. The ways in which Model Composer compiles a model can vary, and depend on settings in the System Generator token. In addition to producing HDL descriptions of hardware, the tool generates auxiliary files. Some files (e.g., project files, constraints files) assist downstream tools, while others (e.g., VHDL test bench) are used for design verification.

Compiling and Simulating Using the System Generator Token	Describes how to use the System Generator token to compile designs into equivalent low-level HDL.
Compilation Results	Describes the low-level files Model Composer produces when **HDL Netlist** is selected on the System Generator token and **Generate** is pushed.
Vivado Project	Describes the example projectModel Composer produces when **HDL Netlist** or **IP Catalog** is selected on the System Generator token and **Generate** is pushed.
HDL Testbench	Describes the VHDL test bench that Model Composer can produce.

Compiling and Simulating Using the System Generator Token

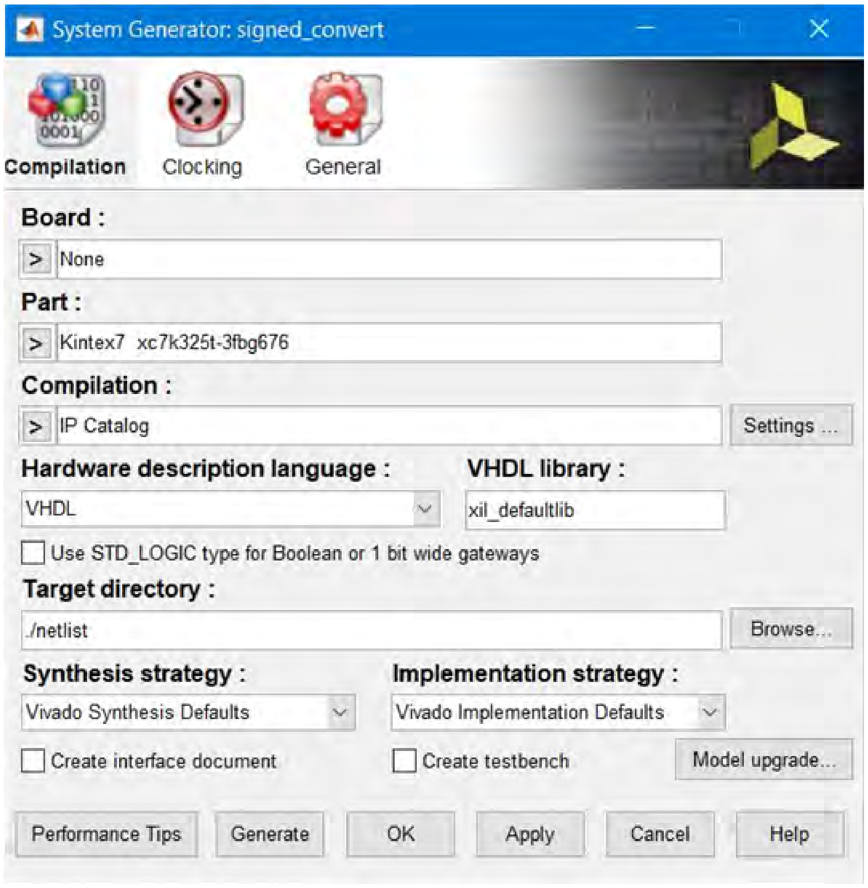
Model Composer automatically compiles designs into low-level representations. Designs are compiled and simulated using the System Generator token. This topic describes how to use the block.

The System Generator token is a member of the Xilinx HDL Blockset's Tools library.

A design must contain at least one System Generator token, but can contain several System Generator tokens on different levels (one per level). A System Generator token that is underneath another in the hierarchy is a *slave*; one that is not a slave is a *master*. The scope of a System Generator token consists of the level of hierarchy into which it is embedded and all Subsystems below that level. Certain parameters (e.g., **Simulink System Period**) can be specified only in a master.

Once a System Generator token is added, it is possible to specify how code generation and synthesis should be handled. The System Generator Token dialog box is shown below:

Figure 22: **System Generator Token Dialog Box**



Board : None

Part : Kintex7 xc7k325t-3fbg676

Compilation : IP Catalog

Hardware description language : VHDL

VHDL library : xil_defaultlib

[ ] Use STD_LOGIC type for Boolean or 1 bit wide gateways

Target directory : ./netlist

Synthesis strategy : Vivado Synthesis Defaults

Implementation strategy : Vivado Implementation Defaults

[ ] Create interface document

[ ] Create testbench

Compilation Type and the Generate Button

Pressing the **Generate** button instructs Model Composer to compile a portion of the design into equivalent low-level results. The portion that is compiled is the sub-tree whose root is the Subsystem containing the block. (To compile the entire design, use a System Generator token placed at the top of the design.) The compilation type (under **Compilation**) specifies the type of result that should be produced. The possible types are:

- **IP Catalog**: Packages the design as an IP core that can be added to the Vivado IP catalog for use in another design.
- **Hardware Co-Simulation (JTAG)**: Generates HW co-simulation library block to verify an algorithm in the hardware.
- **Synthesized Checkpoint**: Creates a design checkpoint file (`synth_1.dcp`) that can then be used in any Vivado IDE project.
- **HDL Netlist**: Generates VHDL or Verilog RTL designs.

Table 1: **System Generator Token Dialog Box Controls**

Control	Description
Board	Specifies a Xilinx, Partner, or Custom board you will use to test your design. For a Partner board or a custom board to appear in the Board list, you must configure Model Composer to access the board files that describe the board. Board awareness in Model Composer is detailed in Specifying Board Support in Model Composer HDL Blockset. When you select a **Board**, the **Part** field displays the name of the Xilinx device on the selected **Board**, and this part name cannot be changed.
Part	Defines the Xilinx part to be used. If you have selected a **Board**, the **Part** field will display the name of the Xilinx device on the selected **Board**, and this part name cannot be changed.
Hardware description language	Specifies the language to be used for HDL netlist of the design. The possibilities are VHDL and Verilog.
VHDL library	Specifies the name of VHDL work library for code generation. The default name is xil_defaultlib.
Use STD_LOGIC type for Boolean or 1 bit wide gateways	If your design's Hardware Description Language (HDL) is VHDL, selecting this option will declare a Boolean or 1-bit port (Gateway In or Gateway Out) as a STD-LOGIC type. If this option is not selected, Model Composer will interpret Boolean or 1-bit ports as vectors.
Target Directory	Defines where Model Composer should write compilation results. Because Model Composer and the FPGA physical design tools typically create many files, it is best to create a separate target directory, i.e., a directory other than the directory containing your Simulink model files. The directory can be an absolute path (e.g. `c:\netlist`) or a path relative to the directory containing the model (e.g. `netlist`).
Synthesis strategy	Choose a Synthesis strategy from the pre-defined strategies in the drop-down list.
Implementation strategy	Choose an Implementation strategy from the pre-defined strategies in the drop-down list.
Create interface document	When this box is checked and the Generate button is activated for netlisting, Model Composer creates an HTM document that describes the design being netlisted. This document is placed in the netlist folder.

Table 1: **System Generator Token Dialog Box Controls** *(cont'd)*

Control	Description
Create testbench	This instructs Model Composer to create an HDL test bench. Simulating the test bench in an HDL simulator compares Simulink simulation results with ones obtained from the compiled version of the design. To construct test vectors, Model Composer simulates the design in Simulink, and saves the values seen at gateways. The top HDL file for the test bench is named `<name>_tb.vhd/.v`, where `<name>` is a name derived from the portion of the design being tested and the extension is dependent on the hardware description language.
FPGA clock period	Defines the period in nanoseconds of the system clock. The value need not be an integer. The period is passed to the Xilinx implementation tools through a constraints file, where it is used as the global PERIOD constraint. Multi-cycle paths are constrained to integer multiples of this value.
Clock pin location	Defines the pin location for the hardware clock. This information is passed to the Xilinx implementation tools through a constraints file.

Simulink System Period

You must specify a value for **Simulink system period** in the System Generator token dialog box. This value tells the underlying rate, in seconds, at which simulations of the design should run. The period must evenly divide all sample periods in the design. For example, if the design consists of blocks whose sample periods are 2, 6, and 8, then the largest acceptable sample period is 2, though other values such as 1 and 0.5 are also acceptable. Sample periods arise in three ways: some are specified explicitly, some are calculated automatically, and some arise implicitly within blocks that involve internal rate changes. For more information on how the system period setting affects the hardware clock, refer to Timing and Clocking.

Before running a simulation or compiling the design, Model Composer verifies that the period evenly divides every sample period in the design.

It is possible to assemble a Model Composer model that is inconsistent because its periods cannot be reconciled. (For example, certain blocks require that they run at the system rate. Driving an up-sampler with such a block produces an inconsistent model.) If, even after updating the system period, Model Composer reports there are conflicts, then the model is inconsistent and must be corrected.

The period control is hierarchical. See Hierarchical Controls for details.

Block Icon Display

The options on this control affect the display of the block icons on the model. After compilation (which occurs when generating, simulating, or by pressing **Ctrl-D**) of the model various information about the block in your model can be displayed, depending on which option is chosen.

- **Default:** Basic information about port directions are shown.
- **Sample rates:** The sample rates of each port are shown like Normalized samples periods and Sample frequencies (MHz).
- **Pipeline stages:** The number of pipeline stages are shown.
- **HDL port names:** the names of the ports are shown
- **Input data types:** The input data types for each port are shown.
- **Output data types:** Output data types for each port are shown.

Hierarchical Controls

The Simulink System Period control (see the topic Simulink System Period above) on the System Generator token is hierarchical. A hierarchical control on a System Generator token applies to the portion of the design within the scope of the token, but can be overridden on other System Generator tokens deeper in the design. For example, suppose Simulink System Period is set in a System Generator token at the top of the design, but is changed in a System Generator token within a Subsystem S. Then that Subsystem will have the second period, but the rest of the design will use the period set in the top level.

Compilation Results

This topic discusses the low-level files Model Composer produces when **HDL Netlist** is selected on the System Generator token and **Generate** is clicked. The files consist of HDL that implements the design. In addition, Model Composer organizes the HDL files, and other hardware files into a Vivado® IDE Project. All files are written to the target directory specified on the System Generator token. If no test bench is requested, then the key files produced by Model Composer are the following:

Table 2: **Compilation Files**

File Name or Type	Description
`<design_name>.vhd/.v`	This file contains a hierarchical structural netlist along with clock/clock enable controls
`<design_name_entity_declarations>.vhd/.v`	This file contains the entity of module definitions of HDL blocks in the design.
`<design_name>.xpr`	This file is the Vivado IDE project file that describes all of the attributes of the Vivado IDE design.

If a test bench is requested, then, in addition to the above, Model Composer produces files that allow simulation results to be compared. The comparisons are between Simulink® simulation results and corresponding results from Questa, or any other RTL simulator supported by Vivado® IDE such as Vivado simulator, or VCS. The additional files are as follows:

Table 3: **Additional Compilation Files**

File Name or Type	Description
Various .dat files	These contain the simulation results from Simulink.
`<design_name>_tb.vhd/.v`	This is a test bench that wraps the design. When simulated, this test bench compares simulation results from the digital simulator against those produced by Simulink.

Using the Constraints File

When a design is compiled during code generation, Model Composer produces *constraints* that tell downstream tools how to process the design. This enables the tools to produce a higher quality implementation, and to do so using considerably less time. Constraints supply the following:

- The period to be used for the system clock.
- The speed, with respect to the system clock, at which various portions of the design must run.
- The pin locations at which ports should be placed.
- The speed at which ports must operate.

The system clock period (i.e., the period of the fastest hardware clock in the design) can be specified in the System Generator token. Model Composer writes this period to the constraints file. Downstream tools use the period as a goal when implementing the design.

Multicycle Path Constraints

Many designs consist of parts that run at different clock rates. For the fastest part, the system clock period is used. For the remaining parts, the clock period is an integer multiple of the system clock period. It is important that downstream tools know what speed each part of the design must achieve. With this information, efficiency and effectiveness of the tools are greatly increased, resulting in reduced compilation times and improved hardware realizations. The division of the design into parts, and the speed at which each part must run, are specified in the constraints file using multicycle path constraints.

IOB Timing and Placement Constraints

When translated into hardware, Model Composer's HDL Gateway In and Gateway Out blocks become input and output ports. The locations of these ports and the speeds at which they must operate can be entered in the Gateway In and Out parameter dialog boxes. Port location and speed are specified in the constraints file by IOB timing.

This topic describes how Model Composer handles hardware clocks in the HDL it generates. Assume the design is named `<design>`, and `<design>` is an acceptable HDL identifier. When Model Composer compiles the design, it writes a collection of HDL entities or modules, the topmost of which is named `<design>`, and is stored in a file named `<design>.vhd/.v`.

The "Clock Enables" Multirate Implementation

Clock and clock enables appear in pairs throughout the HDL. Typical clock names are *clk_1*, *clk_2*, and *clk_3*, and the names of the companion clock enables are *ce_1*, *ce_2*, and *ce_3* respectively. The name tells the rate for the clock/clock enable pair; logic driven by *clk_1* and *ce_1* runs at the system (i.e., fastest) rate, while logic driven by (say) *clk_2* and *ce_2* runs at half the system rate. Clocks and clock enables are not driven in the entity or module named <design> or any subsidiary entities; instead, they are exposed as top-level input ports

The names of the clocks and clock enables in Model Composer HDL suggest that clocking is completely general, but this is not the case. To illustrate this, assume a design has clocks named clk_1 and clk_2, and companion clock enables named *ce_1* and *ce_2* respectively. You might expect that working hardware could be produced if the *ce_1* and *ce_2* signals were tied high, and *clk_2* were driven by a clock signal whose rate is half that of *clk_1*. For most Model Composer designs this does not work. Instead, *clk_1* and *clk_2* must be driven by the same clock, *ce_1* must be tied high, and *ce_2* must vary at a rate half that of *clk_1* and *clk_2*.

IP Instance Caching

For compilation targets that perform Vivado synthesis to generate their output products, Model Composer incorporates a disk cache to speed up the iterative design process.

With the cache enabled for your design, whenever your compilation generates an IP instance for synthesis, and the Vivado synthesis tool creates synthesis output products, the tools create an entry in the cache area.

After the cache is populated, when a new customization of the IP is created which has the exact same properties, the IP is not synthesized again; instead, the cache is referenced and the corresponding synthesis output in the cache is copied to your design's output directory. Because the IP instance is not synthesized again, and this process is repeated for every IP referenced in your design, generation of the output products is completed more quickly.

The following compilation targets invoke Vivado synthesis; these compilation targets will access the cache to synthesize IP in your design.

- Hardware Co-Simulation
- Synthesized Checkpoint

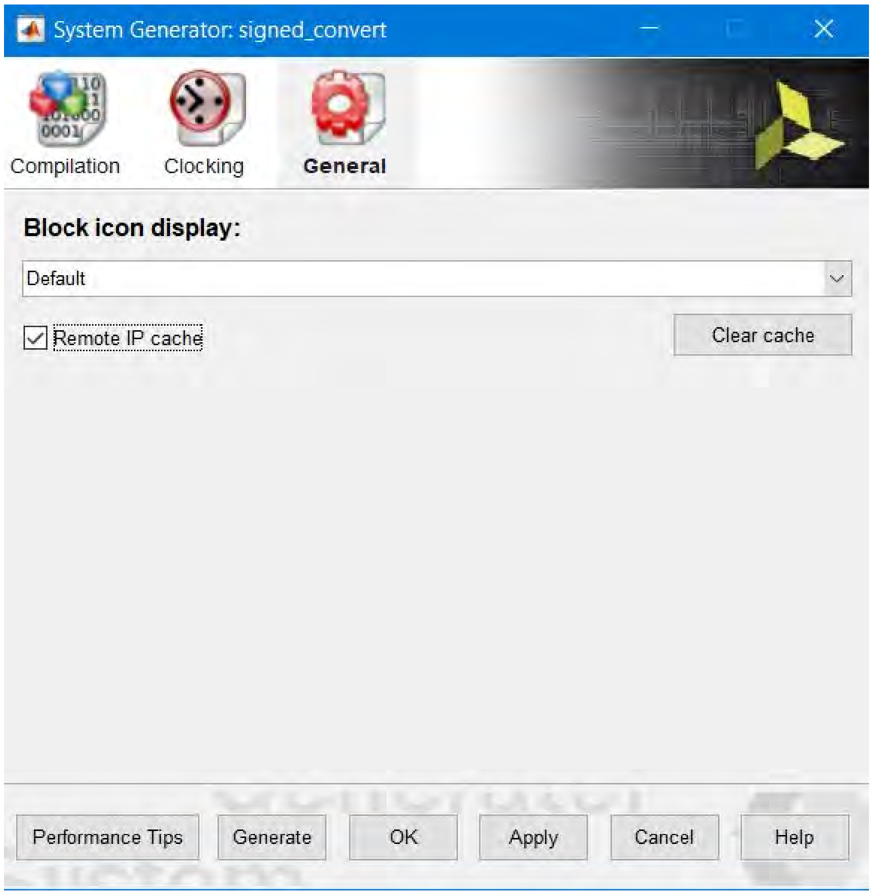
Also, when you compile your design and **Perform analysis** is selected for either **Timing** or **Resource** analysis, Vivado synthesis always runs, regardless of the compilation target. Since timing analysis or resource analysis may be performed several times for a design, enabling IP caching will improve overall performance. For a description of the **Perform analysis** compilation option, see Performing Timing Analysis or Performing Resource Analysis.

The IP cache is shared across multiple Simulink models on your system. If you reuse an IP in one design by including it in another design, and the IP is customized identically and has the same part and language settings in both Simulink models, you can gain the benefit of caching when you compile either of the designs.

You can enable IP caching for your design by selecting **Remote IP cache** in the System Generator token dialog box. The cache will then be referenced for every compilation performed afterwards.

CAUTION! *The IP Cache can grow large, depending on the number of IP present in your design.*

Figure 23: **Block Icon Display**



System Generator: signed_convert

Block icon display: Default

[✓] Remote IP cache

You can clear the cache to save disk space by clicking **Clear cache** in the System Generator token dialog box.

To find the location of the IP cache directory on your system, enter the command `xilinx.environment.getipcachepath` on the MATLAB command line. The full path to the IP cache directory will display in the MATLAB command window.

```
>> xilinx.environment.getipcachepath
ans =
C:/Users/your_id/AppData/Local/Xilinx/Sysgen/SysgenVivado/win64.o/ip
```

IP caching in Model Composer is similar to IP caching in the Vivado Design Suite, described at this link in the *Vivado Design Suite User Guide: Designing with IP* (UG896). However, the IP cache for Model Composer designs is in a different location than the IP cache for Vivado projects.

Vivado Project

The **HDL Netlist** and **IP Catalog** compilation targets also generate an example Vivado project, which represents an integration of the results of Code Generation.

In the case of the **HDL Netlist** compilation target, the Vivado project sets the module designed in Model Composer as the top level and includes instances of IP. Also, if **Create testbench** is selected in the System Generator token, a test bench and stimulus files (`*.dat`) are also added to the project.

In the case of the **IP Catalog** compilation target, an example project is created with the following features:

- The IP generated from Model Composer is already added to the IP catalog associated with the project and available for the RTL flow as well as the IP integrator-based flow.
- The design includes an RTL instantiation of IP called <ip>_0 underneath <design>_stub that indicates how to instanciate such an IP in the RTL flow
- The design includes an RTL test bench called <design>_tb that also instantiates the same IP in the RTL flow.

Note: A test bench is *not* created if **AXI4-Lite** slave interface generation is selected in a Gateway In or Gateway Out block.

- The project also includes an example IP integrator diagram with a Zynq-7000 subsystem if the part selected in this example is a Zynq-7000 SoC part. For all other parts, a MicroBlaze™-based subsystem is created.

HDL Testbench

Ordinarily, Model Composer designs are bit and cycle-accurate, so Simulink simulation results exactly match those seen in hardware. There are, however, times when it is useful to compare Simulink simulation results against those obtained from an HDL simulator. In particular, this makes sense when the design contains black boxes. The **Create Testbench** checkbox in the System Generator token makes this possible.

Suppose the design is named `<design>`, and a System Generator token is placed at the top of the design. Suppose also that in the token the **Compilation** field is set to **HDL Netlist**, and the **Create Testbench** checkbox is selected. When the **Generate** button is clicked, Model Composer produces the usual files for the design, and in addition writes the following:

- A file named `<design>_tb.vhd/.v` that contains a test bench HDL entity.
- Various `.dat` files that contain test vectors for use in an HDL test bench simulation.

You can perform RTL simulation using the Vivado Integrated Design Environment (IDE). For more details, refer to the document *Vivado Design Suite User Guide: Logic Simulation* (UG900).

Model Composer generates the `.dat` files by saving the values that pass through gateways. In the HDL simulation, input values from the `.dat` files are stimuli, and output values are expected results. The test bench is simply a wrapper that feeds the stimuli to the HDL for the design, then compares HDL results against expected ones.

Compiling MATLAB into an FPGA

Model Composer provides direct support for MATLAB through the MCode block. The MCode block applies input values to an M-function for evaluation using Xilinx's fixed-point data type. The evaluation is done once for each sample period. The block is capable of keeping internal states with the use of persistent state variables. The input ports of the block are determined by the input arguments of the specified M-function and the output ports of the block are determined by the output arguments of the M-function. The block provides a convenient way to build finite state machines, control logic, and computation heavy systems.

In order to construct an MCode block, an M-function must be written. The M-file must be in the directory of the model file that is to use the M-file or in a directory in the MATLAB path.

The following text provides examples that use the MCode block:

- Example 1 Simple Selector shows how to implement a function that returns the maximum value of its inputs;
- Example 2 Simple Arithmetic Operations shows how to implement simple arithmetic operations;
- Example 3 Complex Multiplier with Latency shows how to build a complex multiplier with latency;
- Example 4 Shift Operations shows how to implement shift operations;
- Example 5 Passing Parameters into the MCode Block shows how to pass parameters into a MCode block;
- Example 6 Optional Input Ports shows how to implement optional input ports on an MCode block;
- Example 7 Finite State Machines shows how to implement a finite state machine;

- Example 8 Parameterizable Accumulator shows how to build a parameterizable accumulator;
- Example 9 FIR Example and System Verification shows how to model FIR blocks and how to do system verification;
- Example 10 RPN Calculator shows how to model a RPN calculator – a stack machine;
- Example 11 Example of disp Function shows how to use disp function to print variable values.

Simple Selector

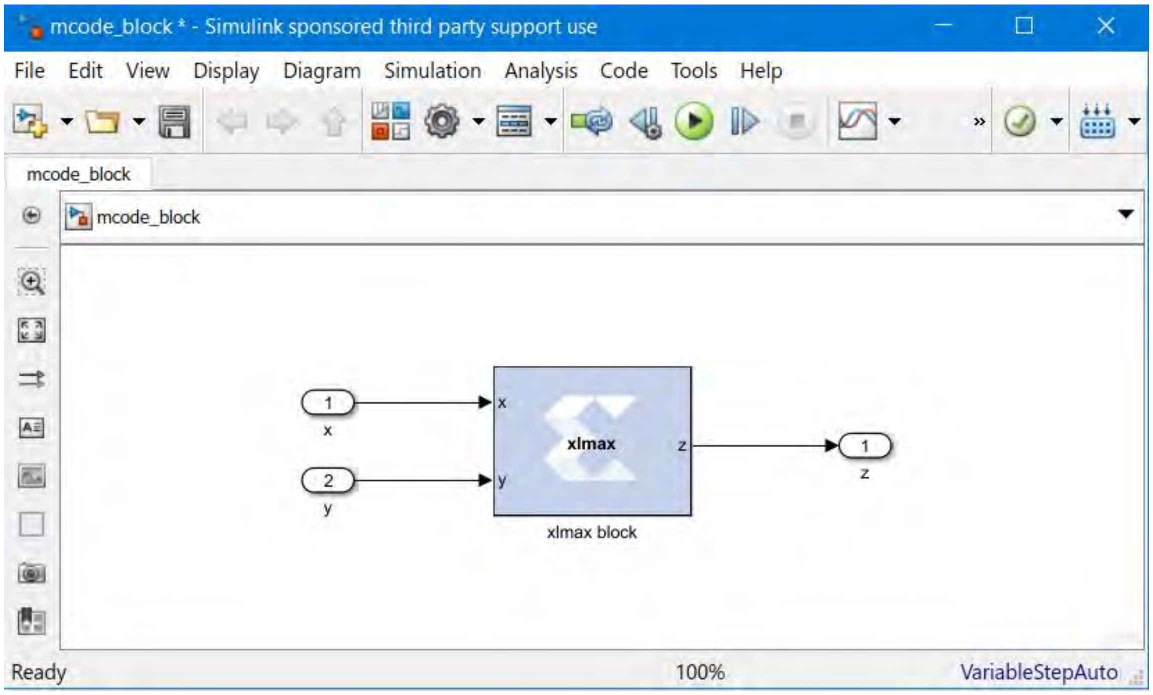
This example is a simple controller for a data path, which assigns the maximum value of two inputs to the output. The M-function is specified as the following and is saved in an M-file `xlmax.m`:

```
function z = xlmax(x, y)
  if x > y
    z = x;
  else
    z = y;
  end
```

The `xlmax.m` file should be either saved in the same directory of the model file or should be in the MATLAB path. Once the `xlmax.m` has been saved to the appropriate place, you should drag a MCode block into your model, open the block parameter dialog box, and enter `xlmax` into the **MATLAB Function** field. After clicking the **OK** button, the block has two input ports x and y, and one output port z.

The following figure shows what the block looks like after the model is compiled. You can see that the block calculates and sets the necessary fixed-point data type to the output port.

Figure 24: **Simple Selector Design**


mcode_block * - Simulink sponsored third party support use
File Edit View Display Diagram Simulation Analysis Code Tools Help
mcode_block
mcode_block
1
x
2
y
x
y
xlmax
z
xlmax block
1
z
Ready
100%
VariableStepAuto

Simple Arithmetic Operations

This example shows some simple arithmetic operations and type conversions. The following shows the `xlSimpleArith.m` file, which specifies the `xlSimpleArith` M-function.

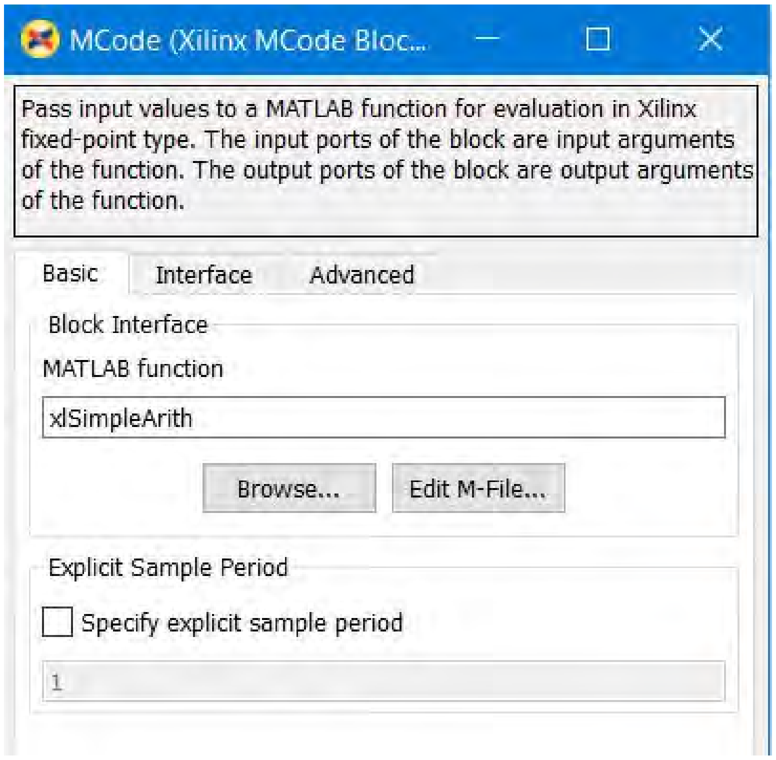
```
function [z1, z2, z3, z4] = xlSimpleArith(a, b)
  % xlSimpleArith demonstrates some of the arithmetic operations
  % supported by the Xilinx MCode block. The function uses xfix()
  % to create Xilinx fixed-point numbers with appropriate
  % container types.%
  % You must use a xfix() to specify type, number of bits, and
  % binary point position to convert floating point values to
  % Xilinx fixed-point constants or variables.
  % By default, the xfix call uses xlTruncate
  % and xlWrap for quantization and overflow modes.
  % const1 is Ufix_8_3
  const1 = xfix({xlUnsigned, 8, 3}, 1.53);
  % const2 is Fix_10_4
  const2 = xfix({xlSigned, 10, 4, xlRound, xlWrap}, 5.687);
  z1 = a + const1;
  z2 = -b - const2;
  z3 = z1 - z2;
  % convert z3 to Fix_12_8 with saturation for overflow
  z3 = xfix({xlSigned, 12, 8, xlTruncate, xlSaturate}, z3);
  % z4 is true if both inputs are positive
  z4 = a>const1 & b>-1;
```

This M-function uses addition and subtraction operators. The MCode block calculates these operations in full precision, which means the output precision is sufficient to carry out the operation without losing information.

One thing worth discussing is the `xfix` function call. The function requires two arguments: the first for fixed-point data type precision and the second indicating the value. The precision is specified in a cell array. The first element of the precision cell array is the type value. It can be one of three different types: `xlUnsigned`, `xlSigned`, or `xlBoolean`. The second element is the number of bits of the fixed-point number. The third is the binary point position. If the element is `xlBoolean`, there is no need to specify the number of bits and binary point position. The number of bits and binary point position must be specified in pair. The fourth element is the quantization mode and the fifth element is the overflow mode. The quantization mode can be one of `xlTruncate`, `xlRound`, or `xlRoundBanker`. The overflow mode can be one of `xlWrap`, `xlSaturate`, or `xlThrowOverflow`. Quanitization mode and overflow mode must be specified as a pair. If the quantization-overflow mode pair is not specified, the `xfix` function uses `xlTruncate` and `xlWrap` for signed and unsigned numbers. The second argument of the `xfix` function can be either a double or a Xilinx fixed-point number. If a constant is an integer number, there is no need to use the `xfix` function. The Mcode block converts it to the appropriate fixed-point number automatically.

After setting the dialog box parameter **MATLAB function** to `xlSimpleArith`, the block shows two input ports a and b, and four output ports z1, z2, z3, and z4.

Figure 25: **xlSimpleArith MCode Parameter**


MCode (Xilinx MCode Bloc...

Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

Basic | Interface | Advanced

Block Interface

MATLAB function: xlSimpleArith

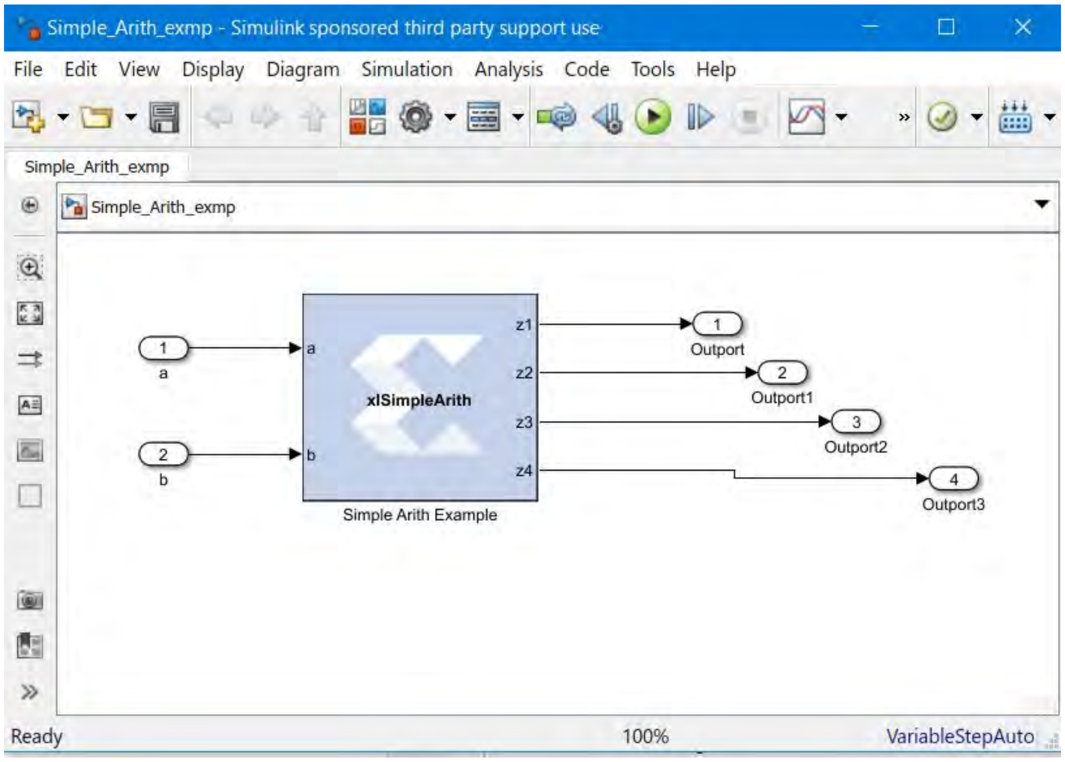
Browse... | Edit M-File...

Explicit Sample Period

☐ Specify explicit sample period

1

Figure 26: **xlSimpleArith Design**



M-functions using Xilinx data types and functions can be tested in the MATLAB Command Window. For example, if you type: `[z1, z2, z3, z4] = xlSimpleArith(2, 3)` in the MATLAB Command Window, you'll get the following lines:

```
UFix(9, 3): 3.500000
Fix(12, 4): -8.687500
Fix(12, 8): 7.996094
Bool: true
```

Notice that the two integer arguments (2 and 3) are converted to fixed-point numbers automatically. If you have a floating-point number as an argument, an `xfix` call is required.

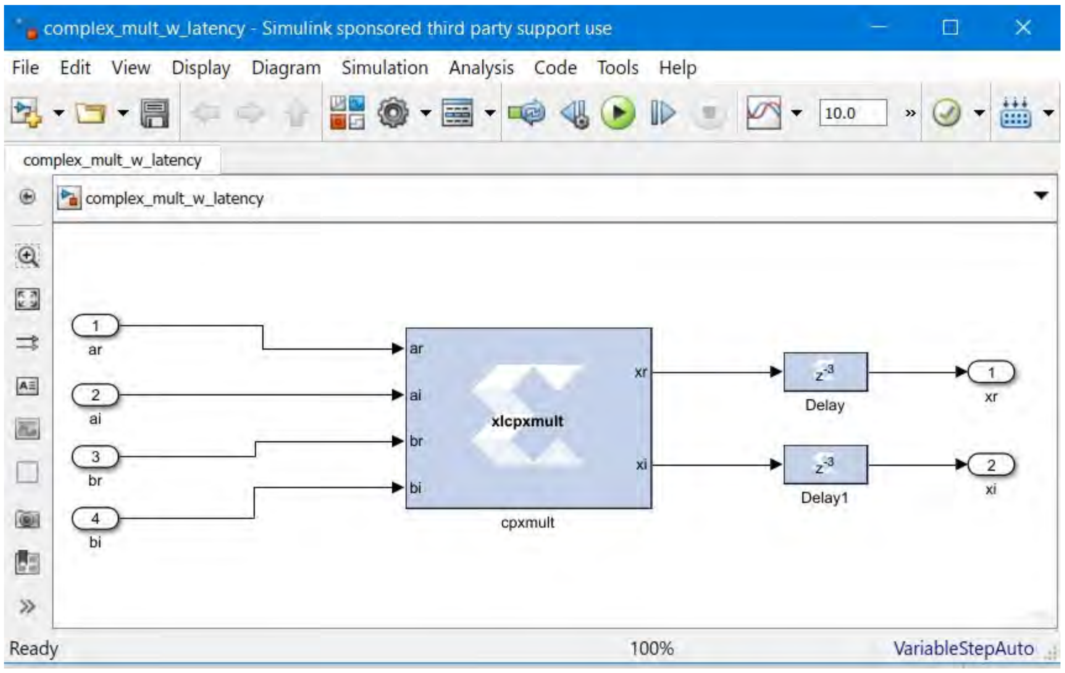
Complex Multiplier with Latency

This example shows how to create a complex number multiplier. The following shows the `xlcpxmult.m` file which specifies the xlcpxmult function.

```
function [xr, xi] = xlcpxmult(ar, ai, br, bi)
  xr = ar * br - ai * bi;
  xi = ar * bi + ai * br;
```

The following diagram shows the sub-system:

Figure 27: **Complex Multiplier Subsystem**



Two delay blocks are added after the MCode block. By selecting the option **Implement using behavioral HDL** on the Delay blocks, the downstream logic synthesis tool is able to perform the appropriate optimizations to achieve higher performance.

Shift Operations

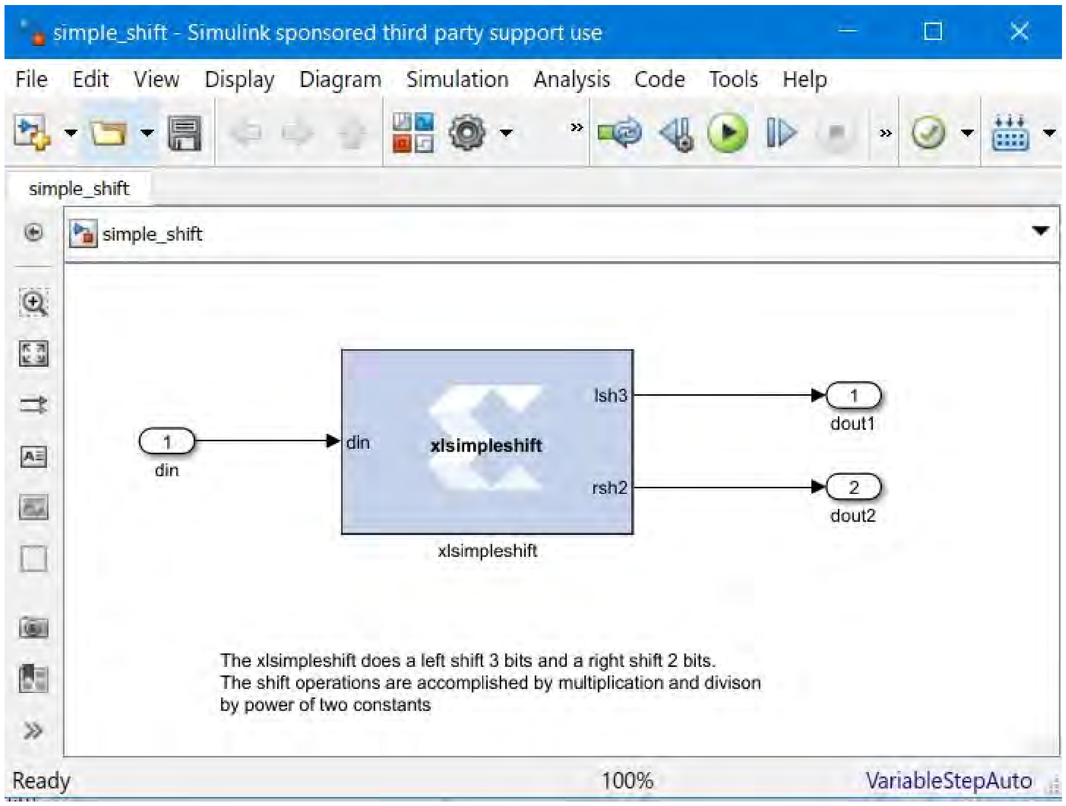
This example shows how to implement bit-shift operations using the MCode block. Shift operations are accomplished with multiplication and division by powers of two. For example, multiplying by 4 is equivalent to a 2-bit left-shift, and dividing by 8 is equivalent to a 3-bit right-shift. Shift operations are implemented by moving the binary point position and if necessary, expanding the bit width. Consequently, multiplying a Fix_8_4 number by 4 results in a Fix_8_2 number, and multiplying a Fix_8_4 number by 64 results in a Fix_10_0 number.

The following shows the `xlsimpleshift.m` file which specifies one left-shift and one right-shift:

```
function [lsh3, rsh2] = xlsimpleshift(din)
  % [lsh3, rsh2] = xlsimpleshift(din) does a left shift
  % 3 bits and a right shift 2 bits.
  % The shift operation is accomplished by
  % multiplication and division of power
  % of two constant.
  lsh3 = din * 8;
  rsh2 = din / 4;
```

The following diagram shows the sub-system after compilation:

Figure 28: **Shift Operations**


simple_shift - Simulink sponsored third party support use
File Edit View Display Diagram Simulation Analysis Code Tools Help
simple_shift
simple_shift
1
din
din
xlsimpleshift
lsh3
rsh2
xlsimpleshift
1
dout1
2
dout2
Ready
100%
VariableStepAuto

The xlsimpleshift does a left shift 3 bits and a right shift 2 bits.
The shift operations are accomplished by multiplication and divison
by power of two constants

Passing Parameters into the MCode Block

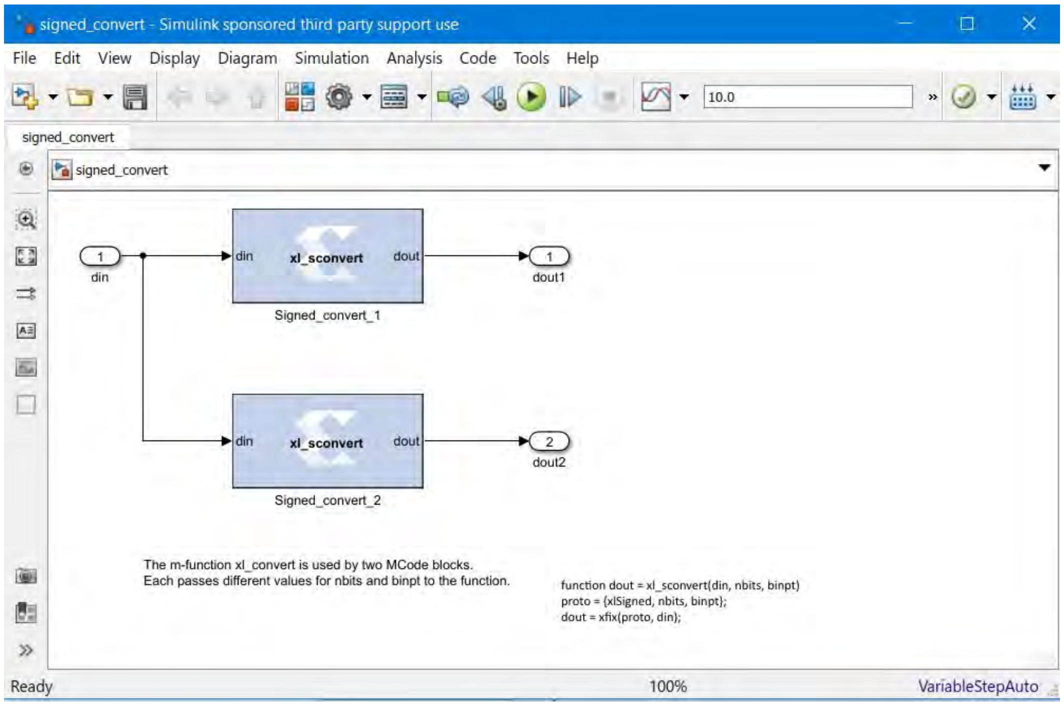
This example shows how to pass parameters into the MCode block. An input argument to an M-function can be interpreted either as an input port on the MCode block, or as a parameter internal to the block.

The following M-code defines an M-function `xl_sconvert` that is contained in file `xl_sconvert.m`:

```
function dout = xl_sconvert(din, nbits, binpt)
  proto = {xlSigned, nbits, binpt};
  dout = xfix(proto, din);
```

The following diagram shows a Subsystem containing two MCode blocks that use M-function `xl_sconvert`. The arguments `nbits` and `binpt` of the M-function are specified differently for each block by passing different parameters to the MCode blocks. The parameters passed to the MCode block labeled `signed convert 1` cause it to convert the input data from type `Fix_16_8` to `Fix_10_5` at its output. The parameters passed to the MCode block labeled `signed convert 2` causes it to convert the input data from type `Fix_16_8` to `Fix_8_4` at its output.

Figure 29: **Subsystem with Two MCode Blocks**

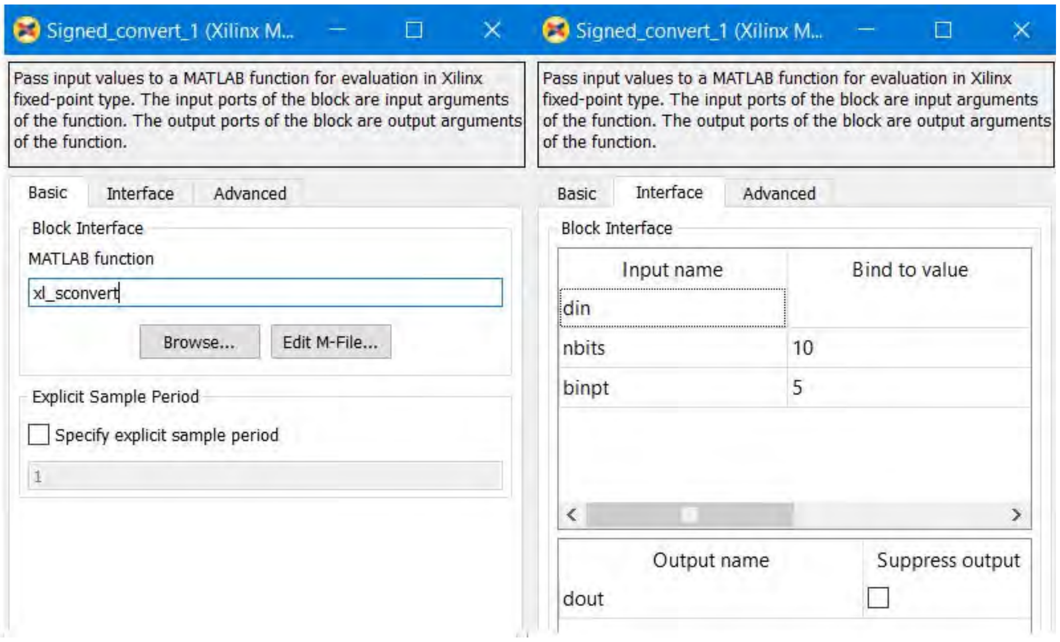

signed_convert - Simulink sponsored third party support use
File Edit View Display Diagram Simulation Analysis Code Tools Help
10.0
signed_convert
din
xl_sconvert
dout
dout1
Signed_convert_1
dout2
Signed_convert_2
Ready
100%
VariableStepAuto

The m-function xl_convert is used by two MCode blocks.
Each passes different values for nbits and binpt to the function.

```
function dout = xl_sconvert(din, nbits, binpt)
proto = {xlSigned, nbits, binpt};
dout = xfix(proto, din);
```

To pass parameters to each MCode block in the diagram above, you can click the **Edit M-File** button on the block GUI then set the values for the M-function arguments. The mask for MCode block `signed convert 1` is shown below:

Figure 30: **Masking MCode Block**



Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

Block Interface

MATLAB function: xl_sconvert

Explicit Sample Period

☐ Specify explicit sample period

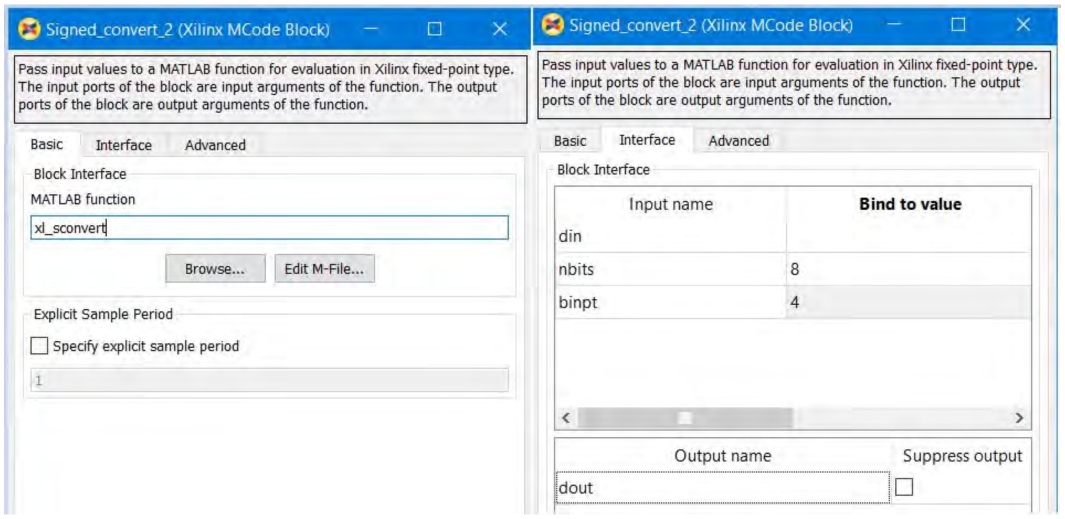
1

Input name	Bind to value
din	
nbits	10
binpt	5

Output name	Suppress output
dout	☐

The above interface window sets the M-function argument `nbits` to be 10 and `binpt` to be 5. The mask for the MCode block `signed convert 2` is shown below:

Figure 31: **Mask for MCode Block Signed Convert 2**


Signed_convert_2 (Xilinx MCode Block)
Basic
Interface
Advanced
Browse...
Edit M-File...
Signed_convert_2 (Xilinx MCode Block)
Basic
Interface
Advanced

Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

Block Interface

MATLAB function: xl_sconvert

Explicit Sample Period

☐ Specify explicit sample period

1

Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

Block Interface

Input name	Bind to value
din	
nbits	8
binpt	4

Output name	Suppress output
dout	☐

Optional Input Ports

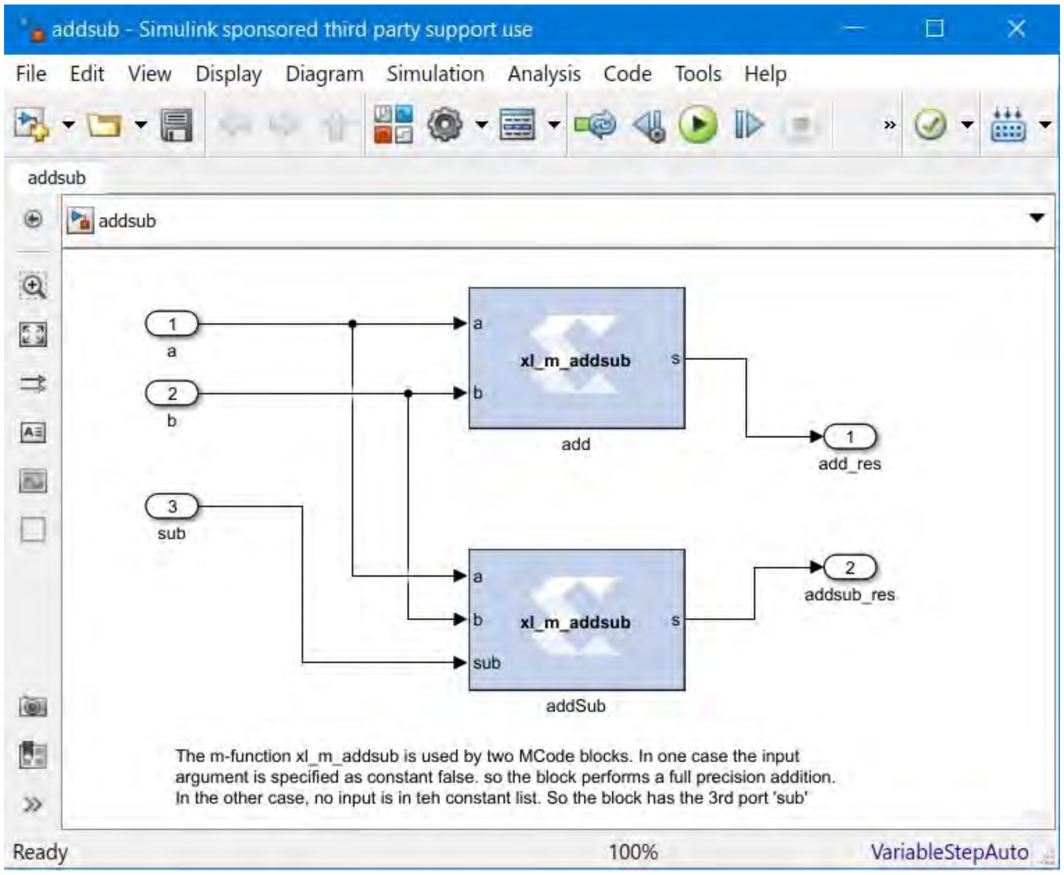
This example shows how to use the parameter passing mechanism of MCode blocks to specify whether or not to use optional input ports on MCode blocks.

The following M-code, which defines M-function `xl_m_addsub` is contained in file `xl_m_addsub.m`:

```
function s = xl_m_addsub(a, b, sub)
  if sub
    s = a - b;
  else
    s = a + b;
  end
```

The following diagram shows a Subsystem containing two MCode blocks that use M-function `xl_m_addsub`.

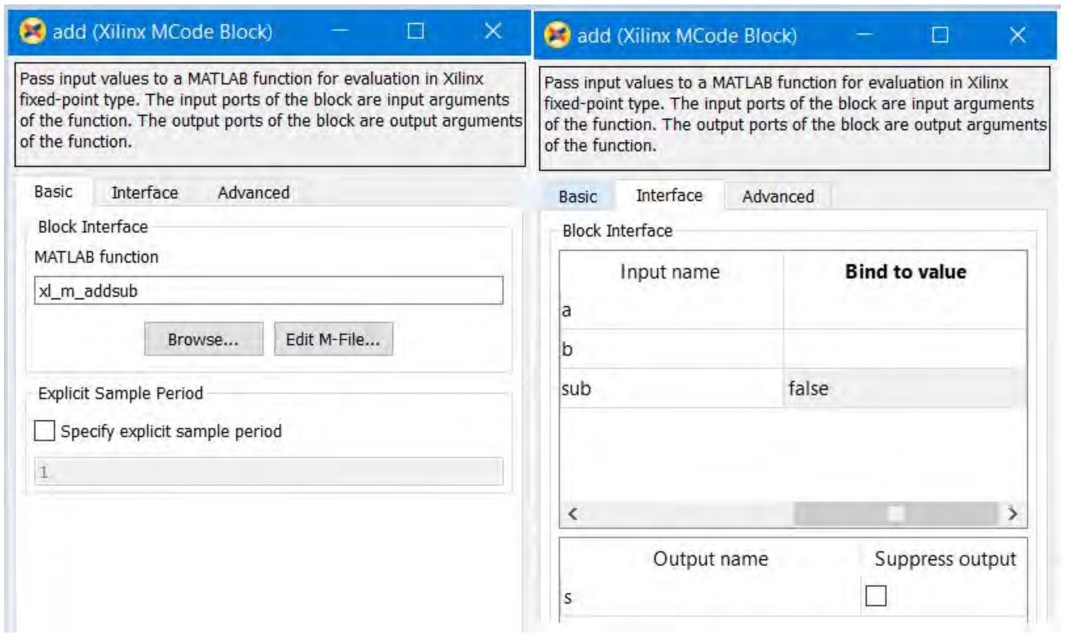
Figure 32: **Two MCode Blocks Using M-Function**


addsub - Simulink sponsored third party support use
File Edit View Display Diagram Simulation Analysis Code Tools Help
addsub
addsub
1
a
2
b
3
sub
a
b
xl_m_addsub
s
add
1
add_res
a
b
sub
xl_m_addsub
s
addSub
2
addsub_res
Ready
100%
VariableStepAuto

The m-function xl_m_addsub is used by two MCode blocks. In one case the input argument is specified as constant false. so the block performs a full precision addition. In the other case, no input is in teh constant list. So the block has the 3rd port 'sub'

The labeled add is shown in below.

Figure 33: **Block Interface Editor of the MCode Block**



add (Xilinx MCode Block)

Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

Basic | Interface | Advanced

Block Interface

MATLAB function: xl_m_addsub

Browse... Edit M-File...

Explicit Sample Period

☐ Specify explicit sample period

1

add (Xilinx MCode Block)

Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

Basic | Interface | Advanced

Block Interface

Input name	**Bind to value**
a	
b	
sub	false

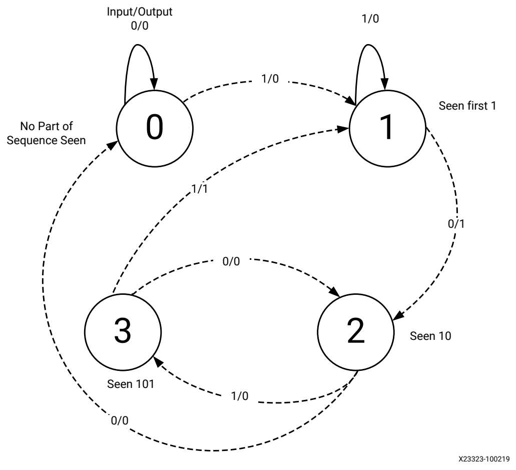
Output name	Suppress output
s	☐

As a result, the `add` block features two input ports `a` and `b`; it performs full precision addition. Input parameter `sub` of the MCode block labeled `addsub` is not bound with any value. Consequently, the `addsub` block features three input ports: `a`, `b`, and `sub`; it performs full precision addition or subtraction based on the value of input port `sub`.

Finite State Machines

This example shows how to create a finite state machine using the MCode block with internal state variables. The state machine illustrated below detects the pattern 1011 in an input stream of bits.

Figure 34: **Finite State Machine Diagram**

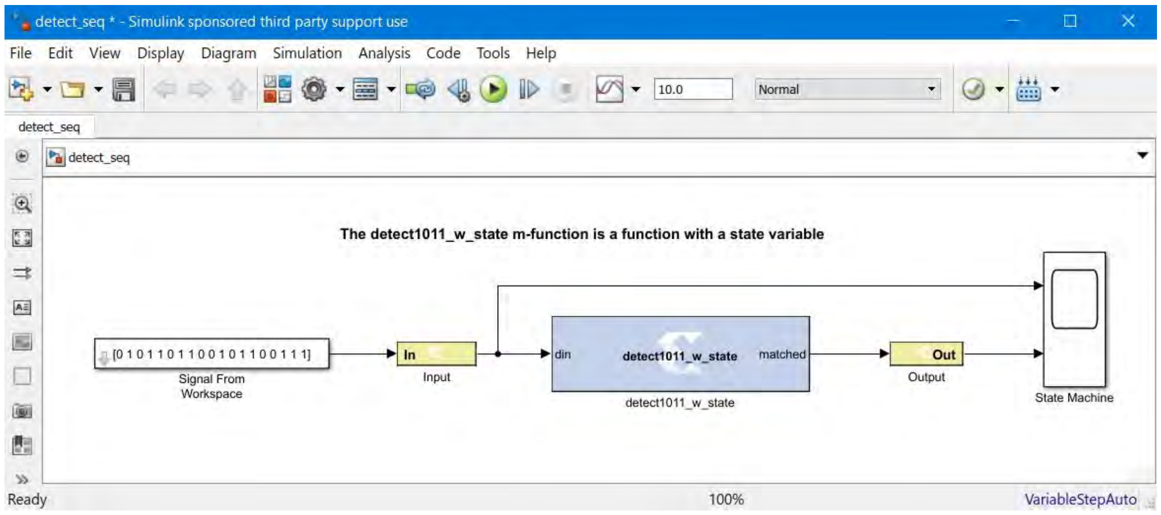


The M-function that is used by the MCode block contains a transition function, which computes the next state based on the current state and the current input. Unlike example 3 though, the M-function in this example defines persistent state variables to store the state of the finite state machine in the MCode block. The following M-code, which defines function `detect1011_w_state` is contained in file `detect1011_w_state.m`:

```
function matched = detect1011_w_state(din)
% This is the detect1011 function with states for detecting a
% pattern of 1011.

  seen_none = 0; % initial state, if input is 1, switch to seen_1
  seen_1 = 1;    % first 1 has been seen, if input is 0, switch
                 % seen_10
  seen_10 = 2;   % 10 has been detected, if input is 1, switch to
                 % seen_1011
  seen_101 = 3;  % now 101 is detected, is input is 1, 1011 is
                 % detected and the FSM switches to seen_1

  % the state is a 2-bit register
  persistent state, state = xl_state(seen_none, {xlUnsigned, 2, 0});

  % the default value of matched is false
  matched = false;

  switch state
    case seen_none
      if din==1
        state = seen_1;
      else
        state = seen_none;
      end
    case seen_1 % seen first 1
      if din==1
        state = seen_1;
      else
        state = seen_10;
      end
    case seen_10 % seen 10
      if din==1
        state = seen_101;
      else
      % no part of sequence seen, go to seen_none
      state = seen_none;
      end
    case seen_101
      if din==1
        state = seen_1;
        matched = true;
      else
        state = seen_10;
        matched = false;
      end
  end
```

The following diagram shows a state machine Subsystem containing a MCode block after compilation; the MCode block uses M-function `detect1101_w_state`.

Figure 35: **Subsystem Containing MCode Block After Compilation**


detect_seq * - Simulink sponsored third party support use
File Edit View Display Diagram Simulation Analysis Code Tools Help
10.0
Normal
detect_seq
The detect1011_w_state m-function is a function with a state variable
[0 1 0 1 1 0 1 1 0 0 1 0 1 1 0 0 1 1 1]
Signal From Workspace
In
Input
din
detect1011_w_state
matched
detect1011_w_state
Out
Output
State Machine
Ready
100%
VariableStepAuto

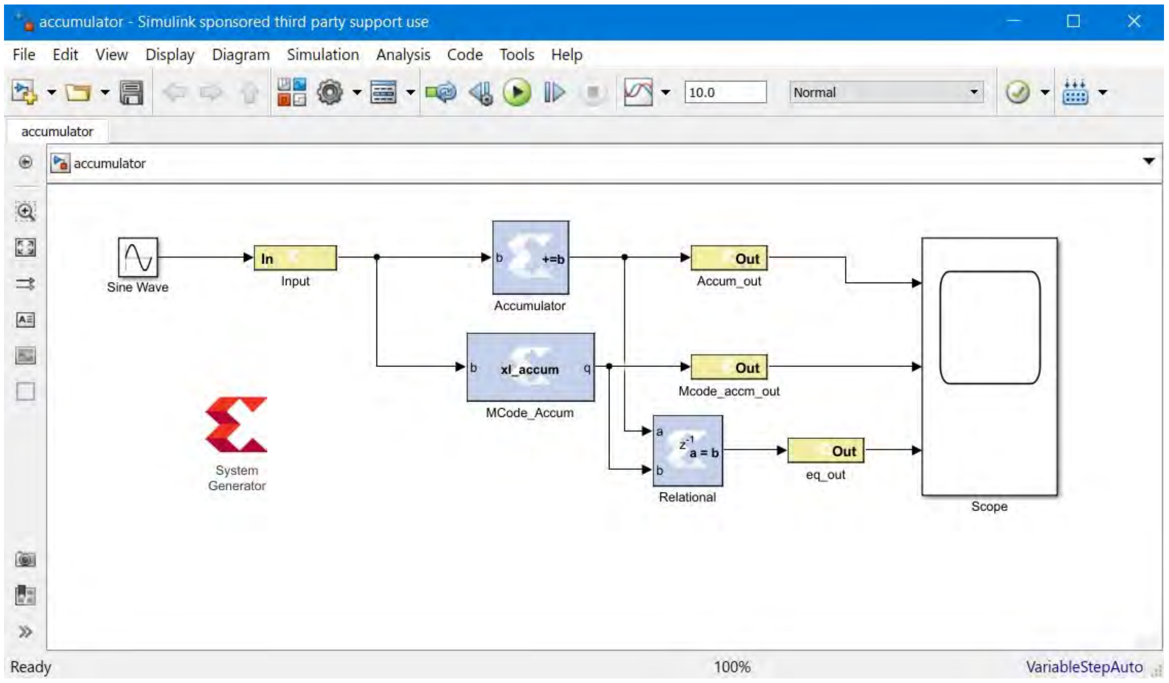
Parameterizable Accumulator

This example shows how to use the MCode block to build an accumulator using persistent state variables and parameters to provide implementation flexibility. The following M-code, which defines function `xl_accum` is contained in file `xl_accum.m`:

```
function q = xl_accum(b, rst, load, en, nbits, ov, op, feed_back_down_scale)
% q = xl_accum(b, rst, nbits, ov, op, feed_back_down_scale) is
% equivalent to our Accumulator block.
  binpt = xl_binpt(b);
  init = 0;
  precision = {xlSigned, nbits, binpt, xlTruncate, ov};
  persistent s, s = xl_state(init, precision);
  q = s;
  if rst
    if load
      % reset from the input port
      s = b;
    else
      % reset from zero
      s = init;
    end
  else
    if ~en
    else
      % if enabled, update the state
      if op==0
        s = s/feed_back_down_scale + b;
      else
        s = s/feed_back_down_scale - b;
      end
    end
  end
```

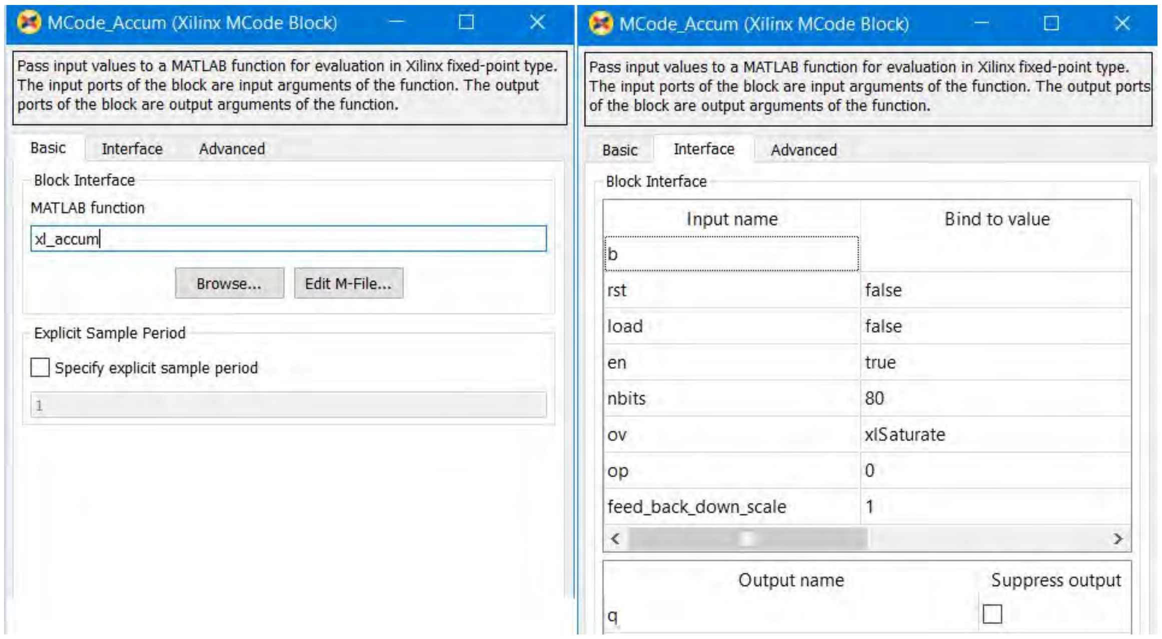
The following diagram shows a Subsystem containing the accumulator MCode block using M-function `xl_accum`. The MCode block is labeled MCode Accumulator. The Subsystem also contains the Xilinx Accumulator block, labeled Accumulator, for comparison purposes. The MCode block provides the same functionality as the Xilinx Accumulator block; however, its mask interface differs in that parameters of the MCode block are specified with a cell array in the Function Parameter Bindings parameter.

Figure 36: **MCode Accumulator**



Optional inputs `rst` and `load` of block `Accum_MCode1` are disabled in the cell array of the Function Parameter Bindings parameter. The block mask for block MCode Accumulator is shown below:

Figure 37: **Mask for MCode Accumulator**



Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

Block Interface

MATLAB function: xl_accum

Explicit Sample Period

☐ Specify explicit sample period

1

Input name	Bind to value
b	
rst	false
load	false
en	true
nbits	80
ov	xlSaturate
op	0
feed_back_down_scale	1

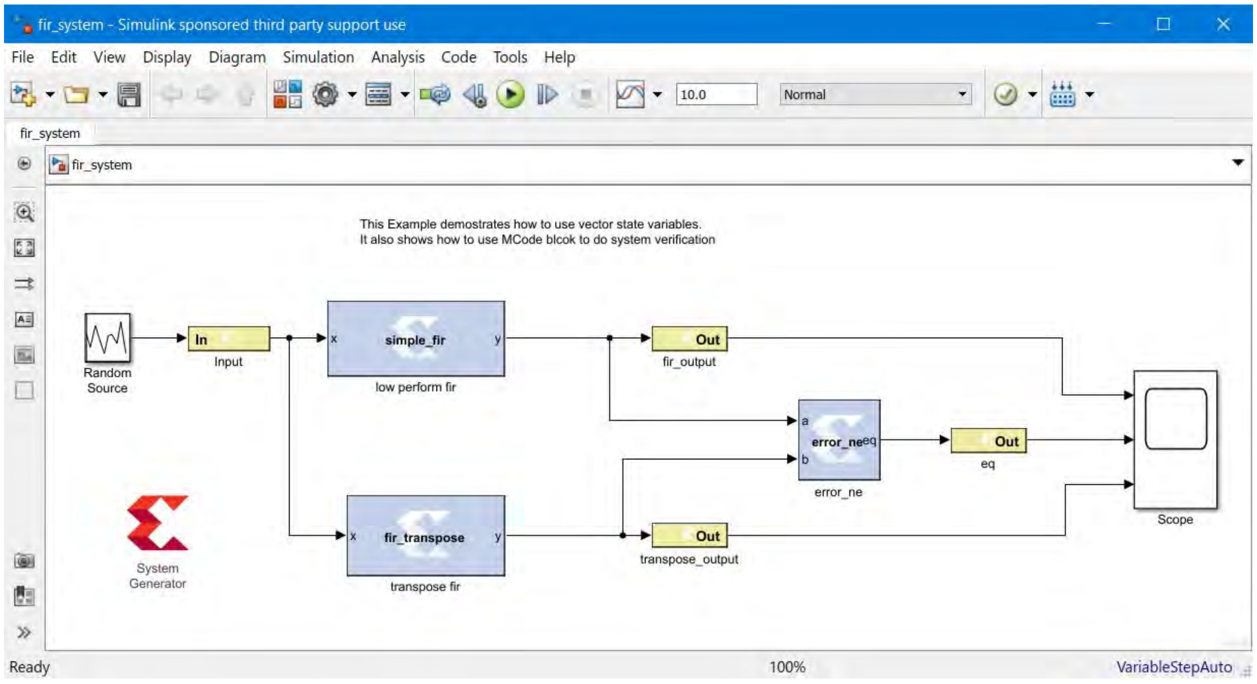
Output name	Suppress output
q	☐

The example contains two additional accumulator Subsystems with MCode blocks using the same M-function, but different parameter settings to accomplish different accumulator implementations.

FIR Example and System Verification

This example shows how to use the MCode block to model FIRs. It also shows how to do system verification with the MCode block.

Figure 38: **FIR Example**

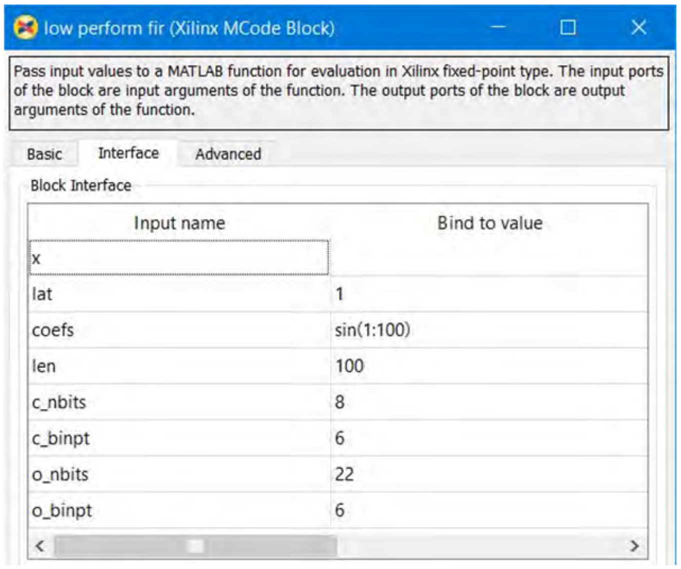

fir_system - Simulink sponsored third party support use
File Edit View Display Diagram Simulation Analysis Code Tools Help
10.0
Normal
fir_system
This Example demostrates how to use vector state variables.
It also shows how to use MCode blcok to do system verification
Random Source
In
Input
x
simple_fir
y
low perform fir
Out
fir_output
a
error_neeq
b
error_ne
Out
eq
Scope
System Generator
x
fir_transpose
y
transpose fir
Out
transpose_output
Ready
100%
VariableStepAuto

The model contains two FIR blocks. Both are modeled with the MCode block and both are synthesizable. The following are the two functions that model those two blocks.

```
function y = simple_fir(x, lat, coefs, len, c_nbits, c_binpt, o_nbits,
o_binpt)
  coef_prec = {xlSigned, c_nbits, c_binpt, xlRound, xlWrap};
  out_prec = {xlSigned, o_nbits, o_binpt};

  coefs_xfix = xfix(coef_prec, coefs);
  persistent coef_vec, coef_vec = xl_state(coefs_xfix, coef_prec);
  persistent x_line, x_line = xl_state(zeros(1, len-1), x);
  persistent p, p = xl_state(zeros(1, lat), out_prec, lat);

  sum = x * coef_vec(0);
  for idx = 1:len-1
    sum = sum + x_line(idx-1) * coef_vec(idx);
    sum = xfix(out_prec, sum);
  end
  y = p.back;
  p.push_front_pop_back(sum);
  x_line.push_front_pop_back(x);
function y = fir_transpose(x, lat, coefs, len, c_nbits, c_binpt, o_nbits,
o_binpt)
  coef_prec = {xlSigned, c_nbits, c_binpt, xlRound, xlWrap};
  out_prec = {xlSigned, o_nbits, o_binpt};
  coefs_xfix = xfix(coef_prec, coefs);
  persistent coef_vec, coef_vec = xl_state(coefs_xfix, coef_prec);
  persistent reg_line, reg_line = xl_state(zeros(1, len), out_prec);
  if lat <= 0
    error('latency must be at least 1');
  end
  lat = lat - 1;
  persistent dly,
  if lat <= 0
    y = reg_line.back;
  else
    dly = xl_state(zeros(1, lat), out_prec, lat);
    y = dly.back;
    dly.push_front_pop_back(reg_line.back);
  end
  for idx = len-1:-1:1
    reg_line(idx) = reg_line(idx - 1) + coef_vec(len - idx - 1) * x;
  end
  reg_line(0) = coef_vec(len - 1) * x;
```

The parameters are configured as following:

Figure 39: **Parameters**



Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

Input name	Bind to value
x	
lat	1
coefs	sin(1:100)
len	100
c_nbits	8
c_binpt	6
o_nbits	22
o_binpt	6

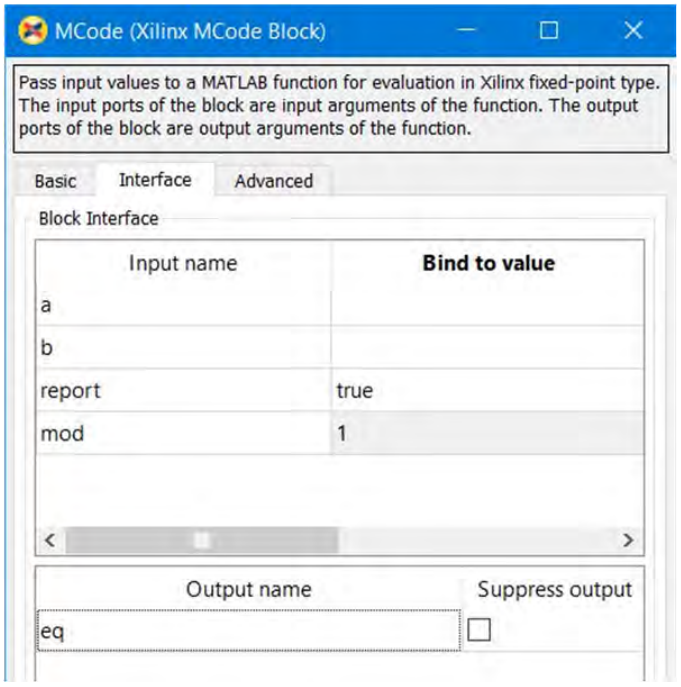
In order to verify that the functionality of two blocks is equal, we also use another MCode block to compare the outputs of two blocks. If the two outputs are not equal at any given time, the error checking block will report the error. The following function does the error checking:

```
function eq = error_ne(a, b, report, mod)
  persistent cnt, cnt = xl_state(0, {xlUnsigned, 16, 0});
  switch mod
    case 1
      eq = a==b;
    case 2
      eq = isnan(a) || isnan(b) || a == b;
    case 3
      eq = ~isnan(a) && ~isnan(b) && a == b;
    otherwise
      eq = false;
    error(['wrong value of mode ', num2str(mod)]);
  end
  if report
```

```
    if ~eq
      error(['two inputs are not equal at time ', num2str(cnt)]);
    end
  end
  cnt = cnt + 1;
```

The block is configured as following:

Figure 40: **Block Configuration**



Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

Block Interface

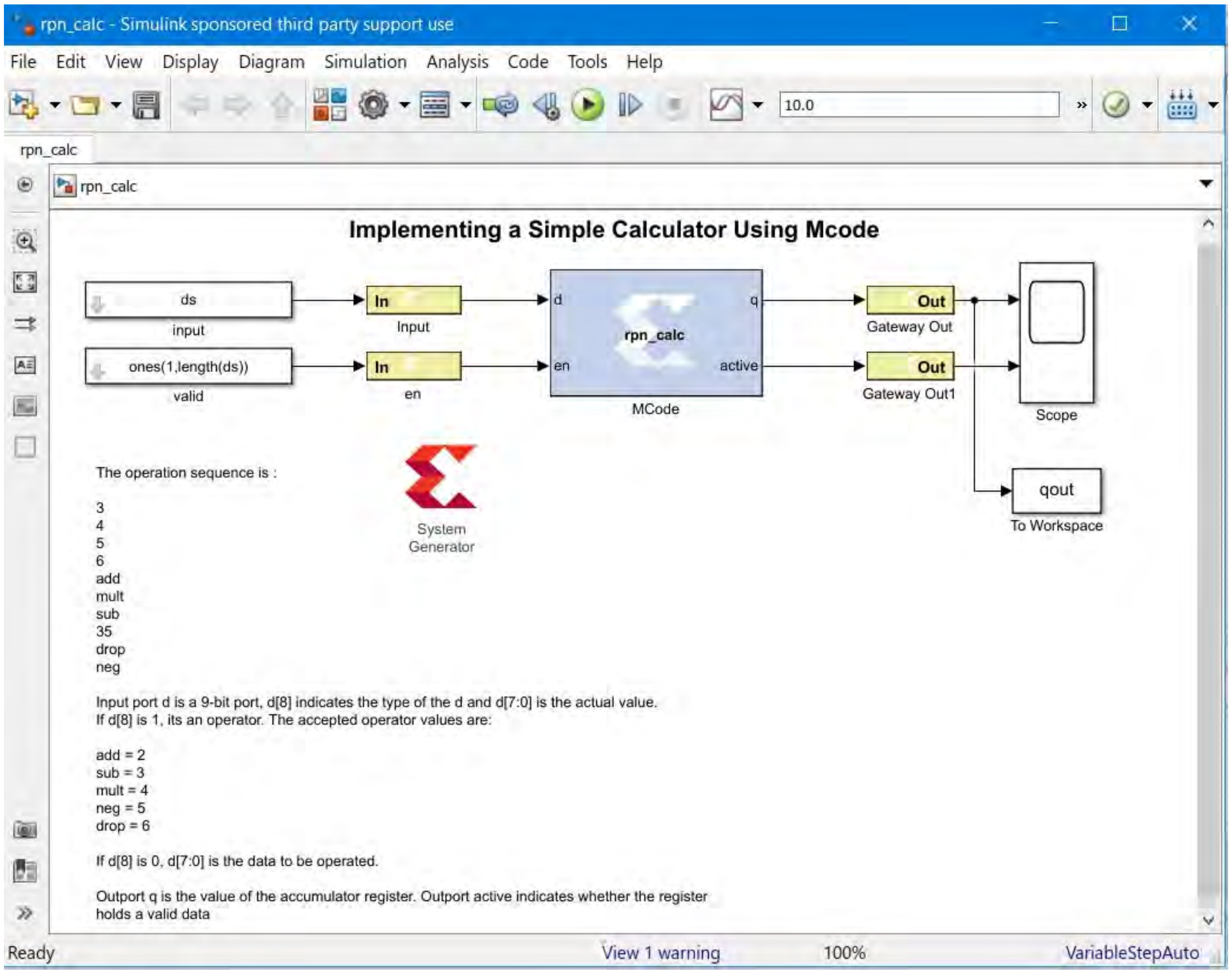
Input name	Bind to value
a	
b	
report	true
mod	1

Output name	Suppress output
eq	☐

RPN Calculator

This example shows how to use the MCode block to model a RPN calculator which is a stack machine. The block is synthesizable:

Figure 41: **RPN Calculator**



Implementing a Simple Calculator Using Mcode

The operation sequence is :

3
4
5
6
add
mult
sub
35
drop
neg

Input port d is a 9-bit port, d[8] indicates the type of the d and d[7:0] is the actual value.
If d[8] is 1, its an operator. The accepted operator values are:

add = 2
sub = 3
mult = 4
neg = 5
drop = 6

If d[8] is 0, d[7:0] is the data to be operated.

Outport q is the value of the accumulator register. Outport active indicates whether the register holds a valid data

The following function models the RPN calculator.

```
function [q, active] = rpn_calc(d, rst, en)
  d_nbits = xl_nbits(d);
  % the first bit indicates whether it's a data or operator
  is_oper = xl_slice(d, d_nbits-1, d_nbits-1)==1;
  din = xl_force(xl_slice(d, d_nbits-2, 0), xlSigned, 0);
  % the lower 3 bits are operator
  op = xl_slice(d, 2, 0);
  % acc the the A register
  persistent acc, acc = xl_state(0, din);
  % the stack is implemented with a RAM and
  % an up-down counter
  persistent mem, mem = xl_state(zeros(1, 64), din);
  persistent acc_active, acc_active = xl_state(false, {xlBoolean});
  persistent stack_active, stack_active = xl_state(false, ...
                                                     {xlBoolean});
  stack_pt_prec = {xlUnsigned, 5, 0};
  persistent stack_pt, stack_pt = xl_state(0, {xlUnsigned, 5, 0});
  % when en is true, it's action
  OP_ADD = 2;
  OP_SUB = 3;
  OP_MULT = 4;
  OP_NEG = 5;
  OP_DROP = 6;
  q = acc;
  active = acc_active;
  if rst
    acc = 0;
    acc_active = false;
    stack_pt = 0;
  elseif en
    if ~is_oper
      % enter data, push
      if acc_active
        stack_pt = xfix(stack_pt_prec, stack_pt + 1);
        mem(stack_pt) = acc;
        stack_active = true;
      else
        acc_active = true;
      end
      acc = din;
    else
      if op == OP_NEG
        % unary op, no stack op
        acc = -acc;
      elseif stack_active
        b = mem(stack_pt);
        switch double(op)
          case OP_ADD
            acc = acc + b;
          case OP_SUB
            acc = b - acc ;
          case OP_MULT
            acc = acc * b;
          case OP_DROP
            acc = b;
        end
        stack_pt = stack_pt - 1;
      elseif acc_active
        acc_active = false;
        acc = 0;
      end
    end
  end
  stack_active = stack_pt ~= 0;
```

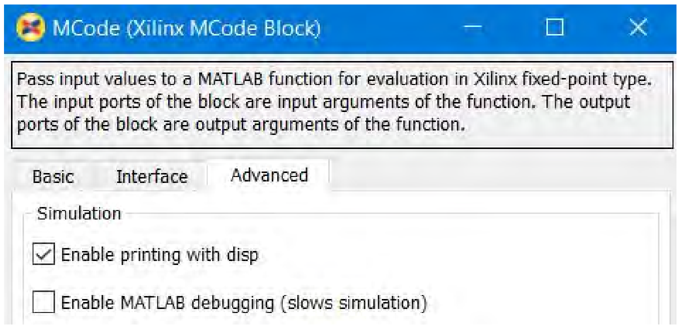
Example of disp Function

The following MCode function shows how to use the disp function to print variable values.

```
function x = testdisp(a, b)
  persistent dly, dly = xl_state(zeros(1, 8), a);
  persistent rom, rom = xl_state([3, 2, 1, 0], a);
  disp('Hello World!');
  disp(['num2str(dly) is ', num2str(dly)]);
  disp('disp(dly) is ');
  disp(dly);
  disp('disp(rom) is ');
  disp(rom);
  a2 = dly.back;
  dly.push_front_pop_back(a);
  x = a + b;
  disp(['a = ', num2str(a), ', ', ...
        'b = ', num2str(b), ', ', ...
        'x = ', num2str(x)]);
  disp(num2str(true));
  disp('disp(10) is');
  disp(10);
  disp('disp(-10) is');
  disp(-10);
  disp('disp(a) is ');
  disp(a);
  disp('disp(a == b)');
  disp(a==b);
```

Check the **Enable printing with disp** option.

Figure 42: **Enable Printing with disp**



Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

[✓] Enable printing with disp

[ ] Enable MATLAB debugging (slows simulation)

Here are the lines that are displayed on the MATLAB console for the first simulation step.

```
mcode_block_disp/MCode (Simulink time: 0.000000, FPGA clock: 0)
Hello World!
num2str(dly) is [0.000000, 0.000000, 0.000000, 0.000000, 0.000000,
0.000000, 0.000000, 0.000000]
disp(dly) is
  type: Fix_11_7,
  maxlen: 8,
  length: 8,
  0: binary 0000.0000000, double 0.000000,
  1: binary 0000.0000000, double 0.000000,
  2: binary 0000.0000000, double 0.000000,
  3: binary 0000.0000000, double 0.000000,
  4: binary 0000.0000000, double 0.000000,
  5: binary 0000.0000000, double 0.000000,
  6: binary 0000.0000000, double 0.000000,
  7: binary 0000.0000000, double 0.000000,
disp(rom) is
  type: Fix_11_7,
  maxlen: 4,
  length: 4,
  0: binary 0011.0000000, double 3.0,
  1: binary 0010.0000000, double 2.0,
  2: binary 0001.0000000, double 1.0,
  3: binary 0000.0000000, double 0.0,
a = 0.000000, b = 0.000000, x = 0.000000
1
disp(10) is
  type: UFix_4_0, binary: 1010, double: 10.0
disp(-10) is
```

```
  type: Fix_5_0, binary: 10110, double: -10.0
disp(a) is
  type: Fix_11_7, binary: 0000.0000000, double: 0.000000
disp(a == b)
  type: Bool, binary: 1, double: 1
```

Importing a Model Composer HDL Design into a Bigger System

A Model Composer design is often a sub-design that is incorporated into a larger HDL design. This topic shows how to embed two Model Composer designs into a larger design and how VHDL created by Model Composer can be incorporated into the simulation model of the overall system.

HDL Netlist Compilation

Selecting the **HDL Netlist** compilation target from the System Generator token instructs Model Composer to generate HDL along with other related files that implement the design. In addition, Model Composer produces auxiliary files that simplify downstream processing such as simulating the design using an Vivado simulator, and performing logic synthesis using Vivado synthesis. See Compilation Types for HDL Library designs for more details.

Integration Design Rules

When a Model Composer model is to be included into a larger design, the following two design rules must be followed.

- **Rule 1:** No Gateway or System Generator token should specify an IOB/CLK location.

Also, IOB timing constraints should be set to "none".

- **Rule 2:** If there are any I/O ports from theModel Composer design that are required to be ports on the top-level design, appropriate buffers should be instantiated in the top-level HDL code.

Configurable Subsystems and Model Composer

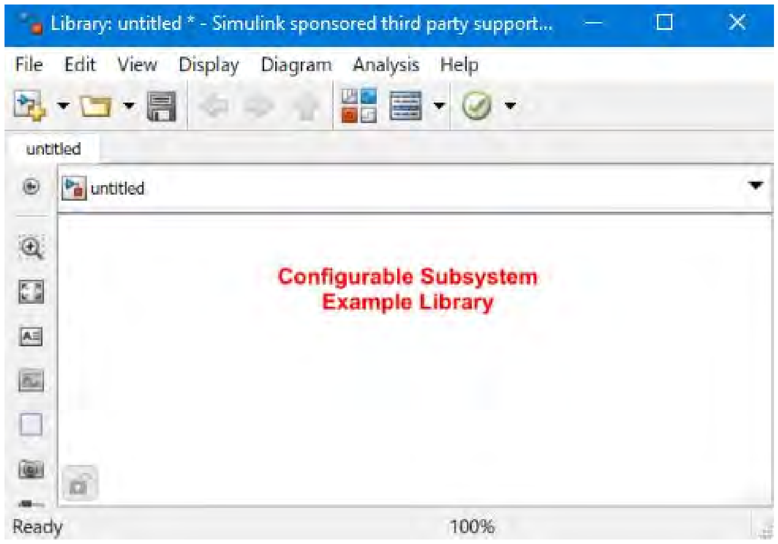
A configurable Subsystem is a kind of block that is made available as a standard part of Simulink. In effect, a configurable Subsystem is a block for which you can specify several underlying blocks. Each underlying block is a possible implementation, and you are free to choose which implementation to use. In Model Composer you might, for example, specify a general-purpose FIR filter as a configurable Subsystem whose underlying blocks are specific FIR filters. Some of the underlying filters might be fast but require much hardware, while others are slow but require less hardware. Switching the choice of the underlying filter allows you to perform experiments that trade hardware cost against speed.

Defining a Configurable Subsystem

A configurable Subsystem is defined by creating a Simulink® library. The underlying blocks that implement a configurable Subsystem are organized in this library. To create such a library, do the following:

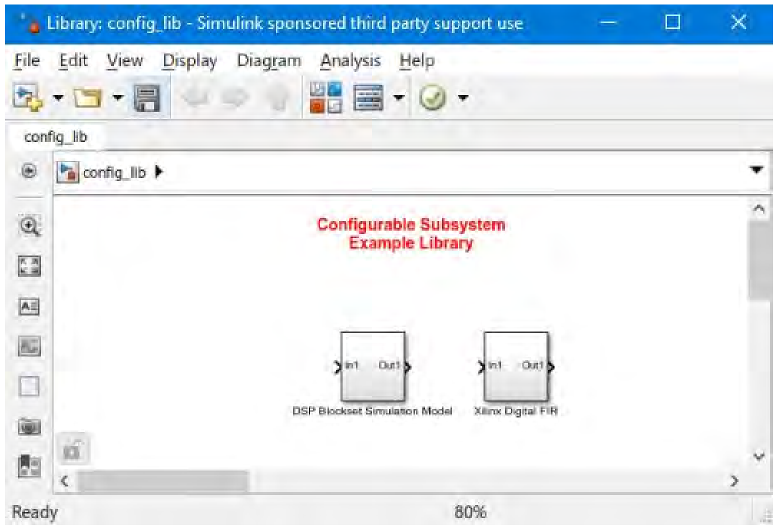
1. Make a new empty library.

Figure 43: **New Empty Library**



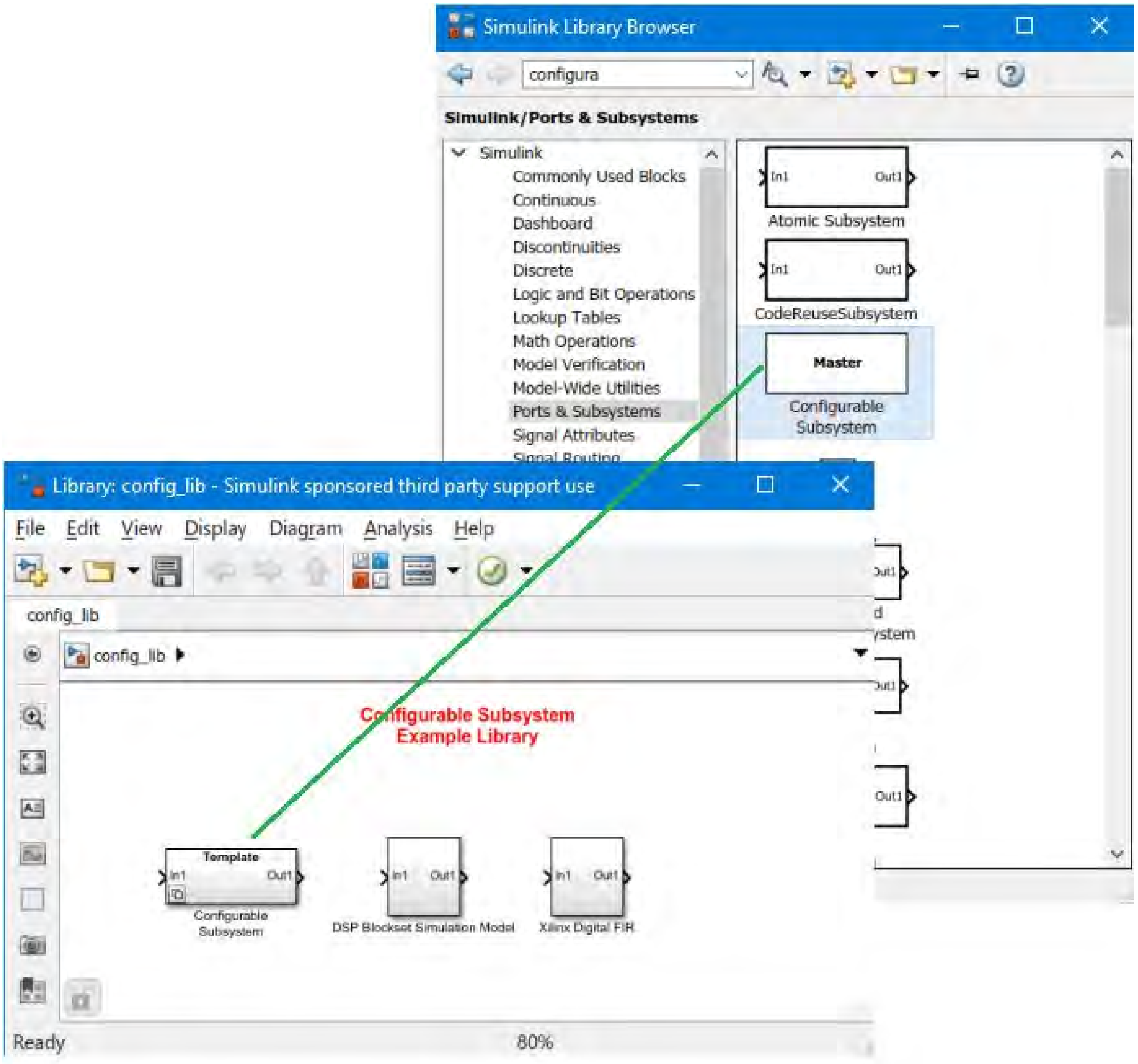
2. Add the underlying blocks to the library.

Figure 44: **Adding Underlying Blocks**



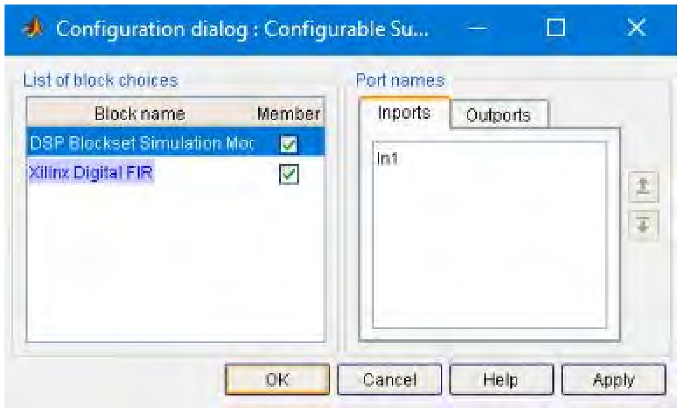
3. Drag a template block into the library. (Templates can be found in the Simulink library browser under Simulink/Ports & Subsystems/Configurable Subsystem.)

Figure 45: **Template**



4. Rename the template block if desired.
5. Save the library.
6. Double-click to open the template for the library.
7. In the template GUI, turn on each check box corresponding to a block that should be an implementation.

Figure 46: **Check Boxes**



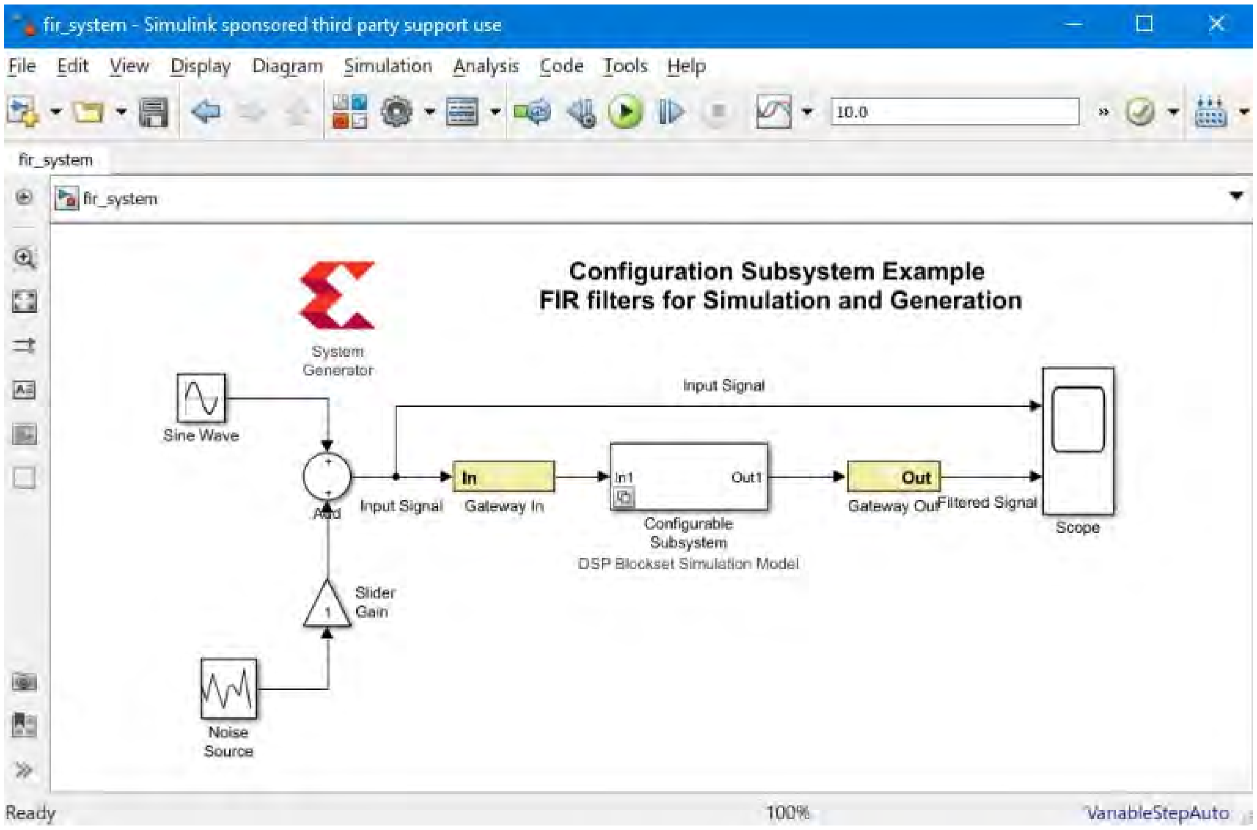
8. Press **OK**, and then save the library again.

Using a Configurable Subsystem

To use a configurable Subsystem in a design, do the following:

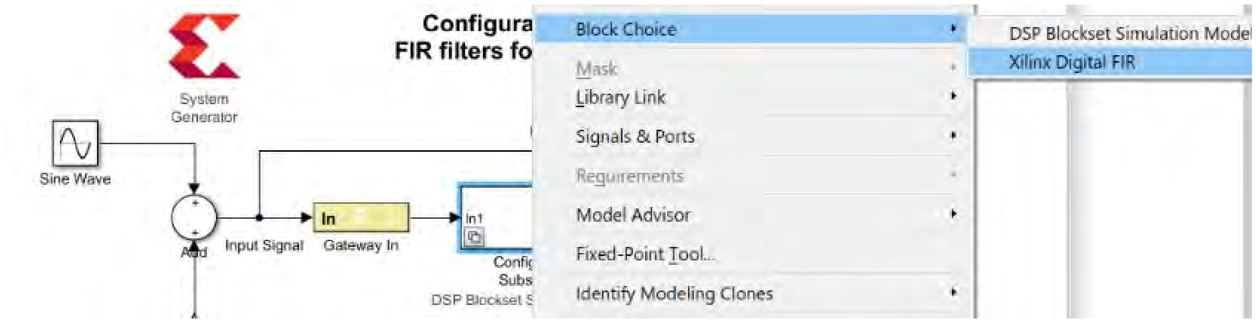
1. As described above, create the library that defines the configurable Subsystem.
2. Open the library.
3. Drag a copy of the template block from the library to the appropriate part of the design.
4. The copy becomes an instance of the configurable Subsystem.

Figure 47: **Configurable Subsystem**


fir_system - Simulink sponsored third party support use
File Edit View Display Diagram Simulation Analysis Code Tools Help
10.0
fir_system
Configuration Subsystem Example
FIR filters for Simulation and Generation
System Generator
Sine Wave
Input Signal
Add
Input Signal
In
Gateway In
In1
Out1
Configurable Subsystem
DSP Blockset Simulation Model
Out
Gateway Out
Filtered Signal
Scope
Slider Gain
Noise Source
Ready
100%
VariableStepAuto

5. Right-click the instance, and under **Block Choice** select the block that should be used as the underlying implementation for the instance.

Figure 48: **Block Choice**



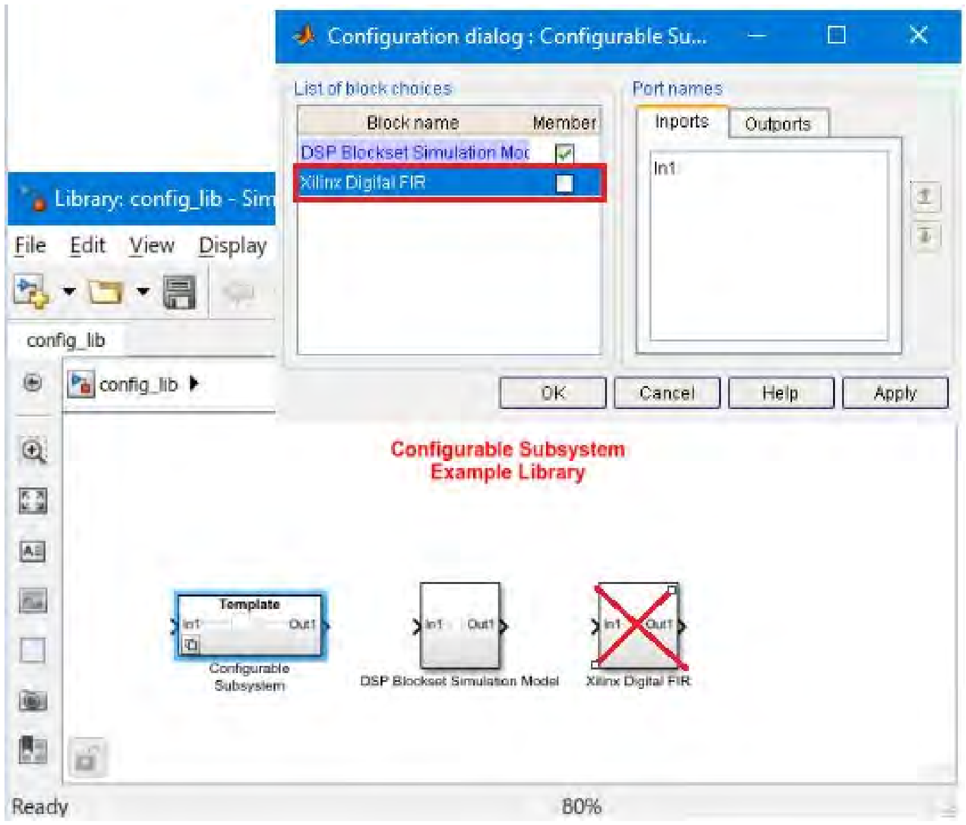
Deleting a Block from a Configurable Subsystem

To delete an underlying block from a configurable Subsystem, do the following:

1. Open and unlock the library for the Subsystem.

2. Double-click the template, and deselect the checkbox associated with the block to be deleted.
3. Press **OK**, and then delete the block.

Figure 49: **Deleting a Block**



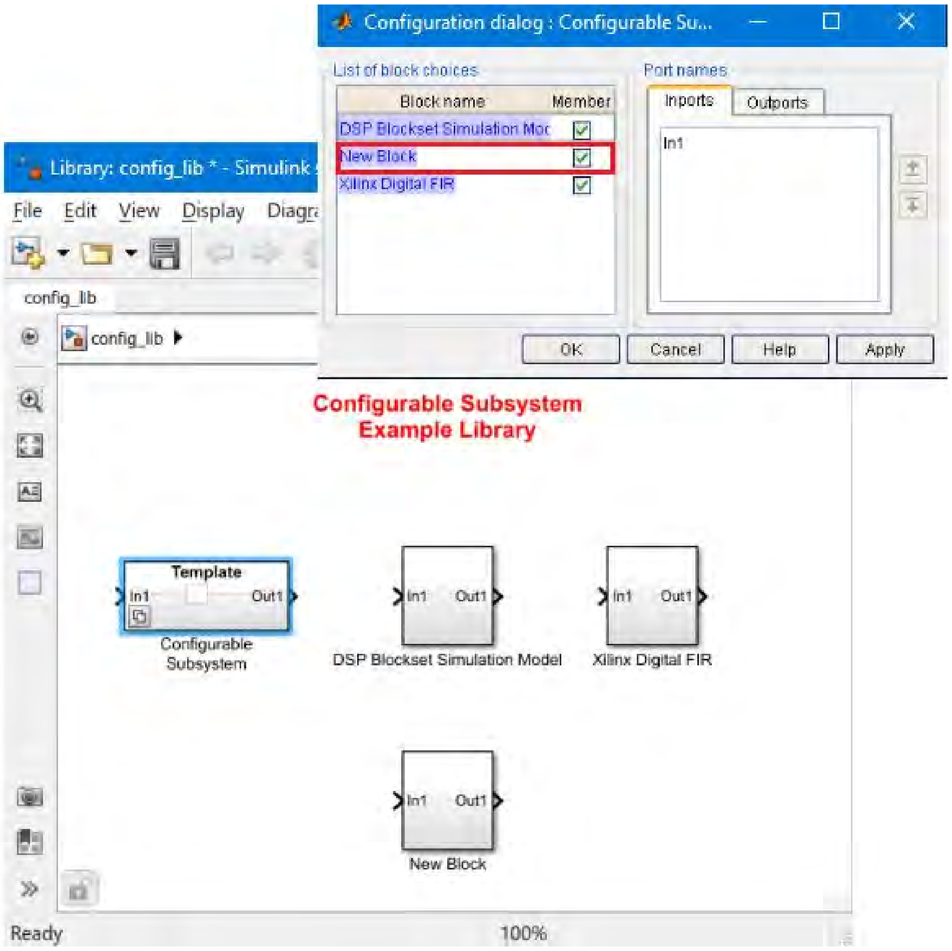
4. Save the library.
5. Compile the design by typing **Ctrl + D**.
6. If necessary, update the choice for each instance of the configurable Subsystem.

Adding a Block to a Configurable Subsystem

To add an underlying block to a configurable Subsystem, do the following:

1. Open and unlock the library for the Subsystem.
2. Drag a block into the library.
3. Double-click the template, and select the checkbox next to the added block.

Figure 50: **Add a Block**



4. Press **OK**, and then save the library.
5. Compile the design by typing **Ctrl-D**.
6. If necessary, update the choice for each instance of the configurable Subsystem.

Notes for Higher Performance FPGA Design

If you focus all your optimization efforts using the back-end implementation tools, you may not be able to achieve timing closure because of the following reasons:

- The more complex IP blocks in a Model Composer design like FIR Compiler and FFT are generated under the hood. They are provided as highly-optimized netlists to the synthesis tool and the implementation tools, so further optimization may not be possible.
- Model Composer netlisting produces HDL code with many instantiated primitives such as registers, BRAMs, and DSP48E1s. There is not much a synthesis tool can do to optimize these elements.

The following tips focus on what you can do in Model Composer to increase the performance of your design before you start the implementation process.

- Review the Hardware Notes Included with Each Block Dialog Box
- Register the Inputs and Outputs of Your Design
- Insert Pipeline Registers
- Use Saturation Arithmetic and Rounding Only When Necessary
- Set the Data Rate Option on All Gateway Blocks
- Pipeline for Maximum Performance
- Other Things to Try

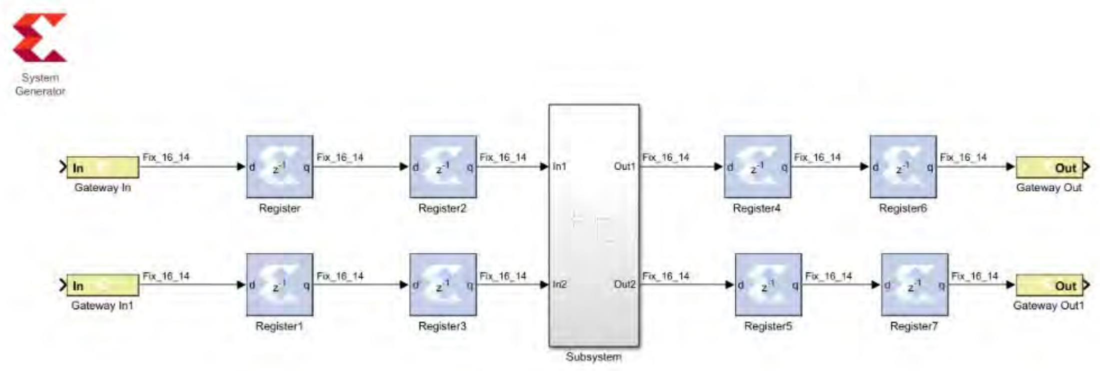
Review the Hardware Notes Included with Each Block Dialog Box

Pay close attention to the Hardware Notes included in the block dialog boxes. Many blocks in the Xilinx Blockset library have notes that explain how to achieve the most hardware efficient implementation. For example, the notes point out that the Scale block costs nothing in hardware. By contrast, the Shift block (which is sometimes used for the same purpose) can use hardware.

Register the Inputs and Outputs of Your Design

Register the inputs and outputs of your design. As shown below, this can be done by placing one or more Delay blocks with a latency 1 or Register blocks after the Gateway In and before Gateway Out blocks. Selecting any of the Register block features adds hardware.

Figure 51: **Register Inputs and Outputs**

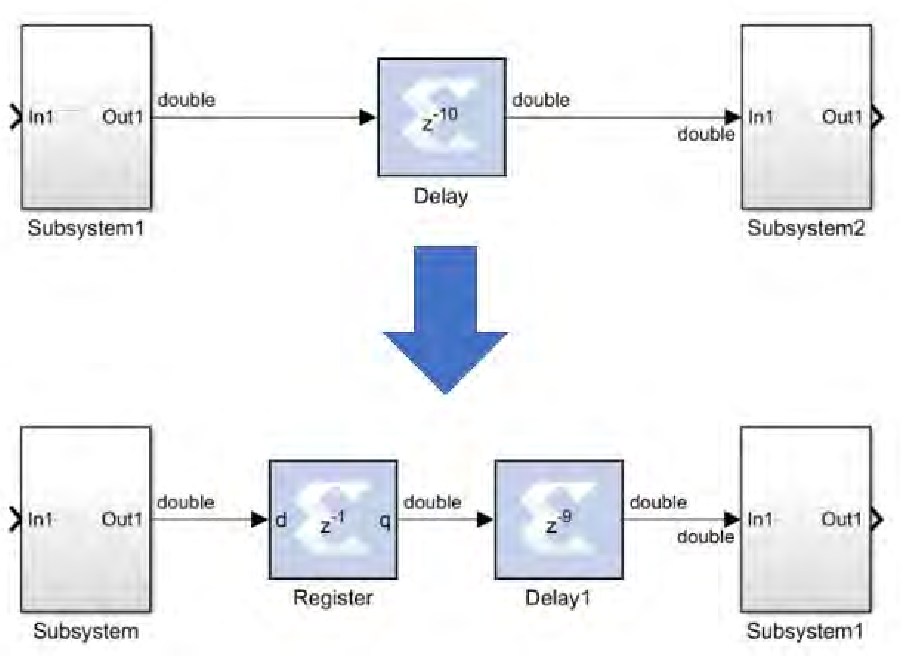


Double registering the I/Os may also be beneficial. This can be performed by instantiating two separate Register blocks, or by instantiating two Delay blocks, each having latency 1. This allows one of the registers to be packed into the IOB and the other to be placed next to the logic in the FPGA fabric. A Delay block with latency 2 does not give the same result because the block with a latency of 2 is implemented using an SRL32 and cannot be packed into an IOB.

Insert Pipeline Registers

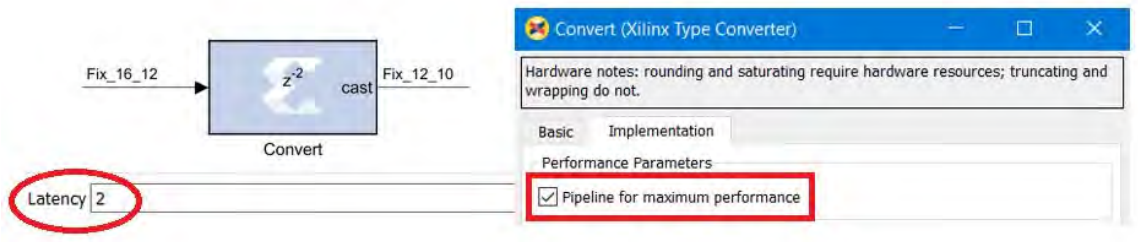
Insert pipeline registers wherever possible and reasonable. Deep pipelines are efficiently implemented with the Delay blocks since the SRL32 primitive is used. If an initial value is needed on a register, the Register block should be used. Also, if the input path of an SRL32 is failing timing, you should place a Register block before the related Delay block and reduce the latency of the Delay block by one. This allows the router more flexibility to place the Register and Delay block (SRL + Register) away from each other to maximize the margin for the routing delay of this path.

Figure 52: **Pipeline Registers**



As shown in the following figure, the Convert block can be pipelined with embedded register stages to guarantee maximum performance.

Figure 53: **Convert Block**

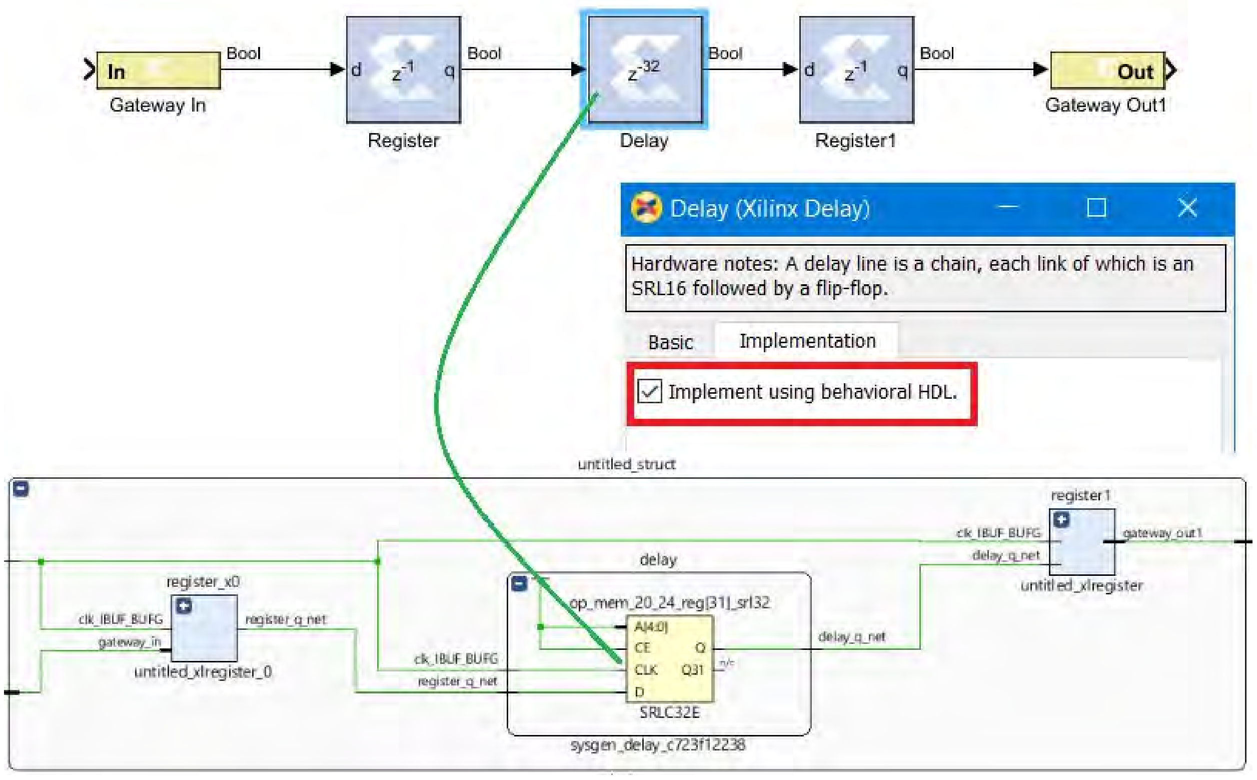


Hardware notes: rounding and saturating require hardware resources; truncating and wrapping do not.

☑ Pipeline for maximum performance

To achieve a more efficient implementation on some Xilinx blocks, you can select the **Implement using behavioral HDL** option. As shown below, if the delay on a Delay block is 32 or greater, Xilinx synthesis infers a SRLC32E (32-bit Shift-Register) which maps into a single LUT.

Figure 54: **Implement Using Behavioral HDL**



Delay (Xilinx Delay)

Hardware notes: A delay line is a chain, each link of which is an SRL16 followed by a flip-flop.

Basic | Implementation

☑ Implement using behavioral HDL.

For block RAMs (BRAMs), use the internal output register. You do this by setting the latency from 1 (the default) to 2. This enables the block RAM output register.

When you are using DSP48E1s, use the input, output and internal registers; for FIFOs, use the embedded registers option. Also, check all the high-level IP blocks for pipelining options.

Use Saturation Arithmetic and Rounding Only When Necessary

Saturation arithmetic and rounding have area and performance costs. Use only if necessary. For example a Reinterpret block doesn‘t cost any logic. A Convert (cast) block doesn‘t cost any logic if Quantization is set to Truncate and if Overflow is set to Wrap. If the data type requires the use of the Rounding and Saturation options, then pipeline the Convert block with embedded register stages. If you are using a DSP48E1, the rounding can be done within the DSP48E1.

Set the Data Rate Option on All Gateway Blocks

Select the IOB timing constraint option **Data Rate** on all **Gateway In** and **Gateway Out** blocks. When **Data Rate** is selected, the IOBs are constrained at the data rate at which the IOBs operate. The rate is determined by the **Simulink system period (sec)** field in the System Generator token and the sample rate of the Gateway relative to the other sample periods in the design.

Pipeline for Maximum Performance

For Model Composer HDL blocks that use Xilinx LogiCORE™ IP internally, the default tool behavior is to place at least one register outside of the core. For latency values greater than the optimum value of the core, the optimal pipeline registers are placed inside the core, and the remainder of the registers get pushed out.

Other Things to Try

- Change the Source Design
  - Use Additional Pipelining

    Use the Output and Pipeline registers inside block RAM and DSP48s.
  - Run Functions in Parallel

    Run functions in parallel at a slower clock rate
  - Use Retiming Techniques

    Move existing registers through combinational logic.
  - Use Hard Cores where Possible

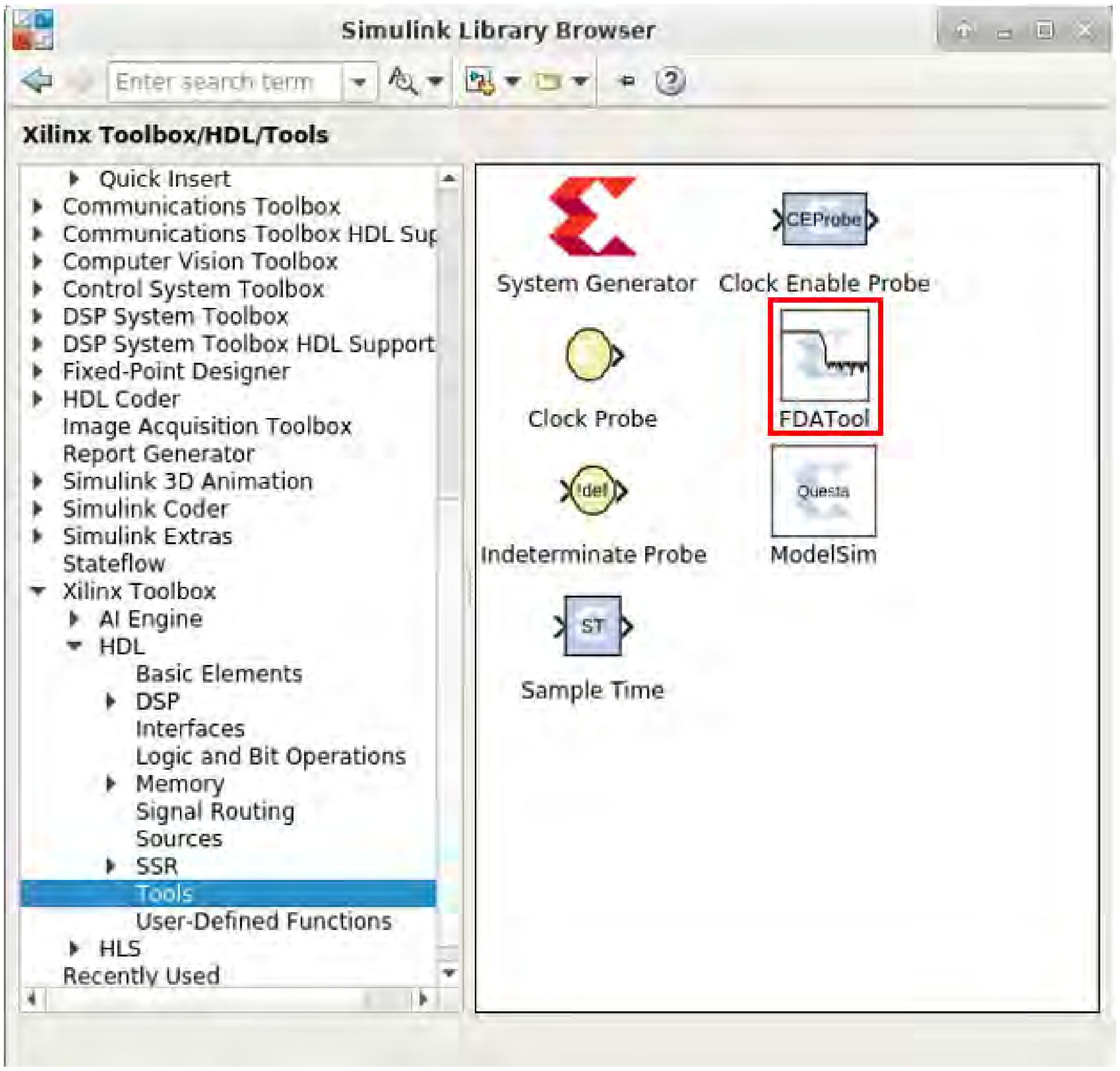
    Use Block RAM instead of distributed RAM.
  - Use a Different Design Approach for Functions
- Avoid Over-Constraining the Design

  Do not over-constrain the design and use up/down sample blocks where appropriate.
- Consider Decreasing the Frequency of Critical Design Modules
- Squeeze Out the Implementation Tools
  - Try Different Synthesis Options.
  - Floorplan Critical Modules

Using the FDATool in Digital Filter Applications

The FDATool block is used to define the filter order and coefficients, and the HDL blocks are used to implement a filter. The Tools library in the HDL Blockset contains the FDATool block.

Figure 55: **FDATool Block**



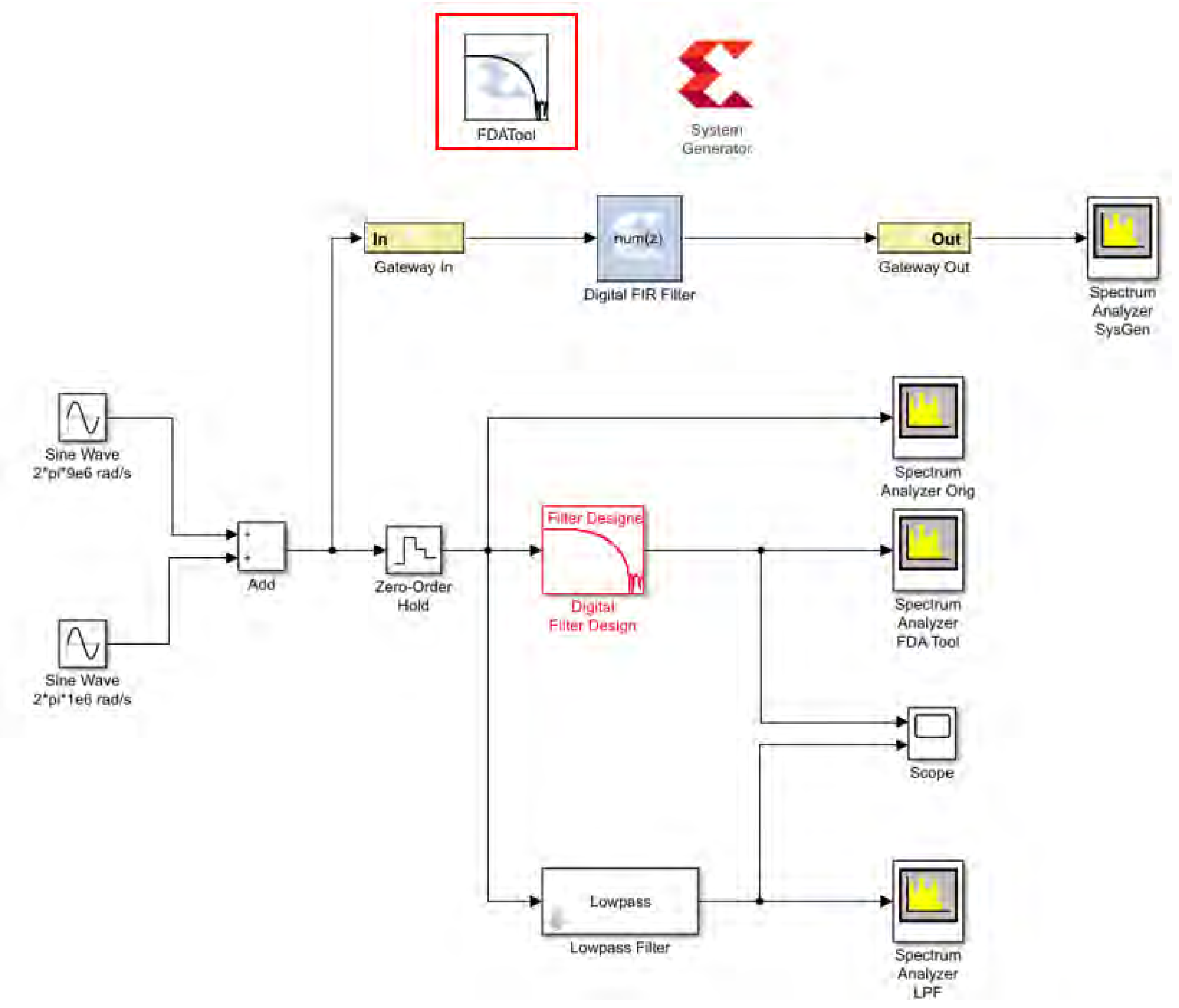
A simple Model Composer model below illustrates a standard FIR filter design using the FDATool and digital FIR filter block.

The design uses two sine wave sources which are being added together and passed separately through two low-pass filters.

- The first filter is the one that could be implemented using the Xilinx HDL blockset. It is a digital low pass filter implemented using the Digital FIR filter block.
- The second filter is what is referred to as a reference filter. A low pass filter is implemented using a Direct-form FIR structure.

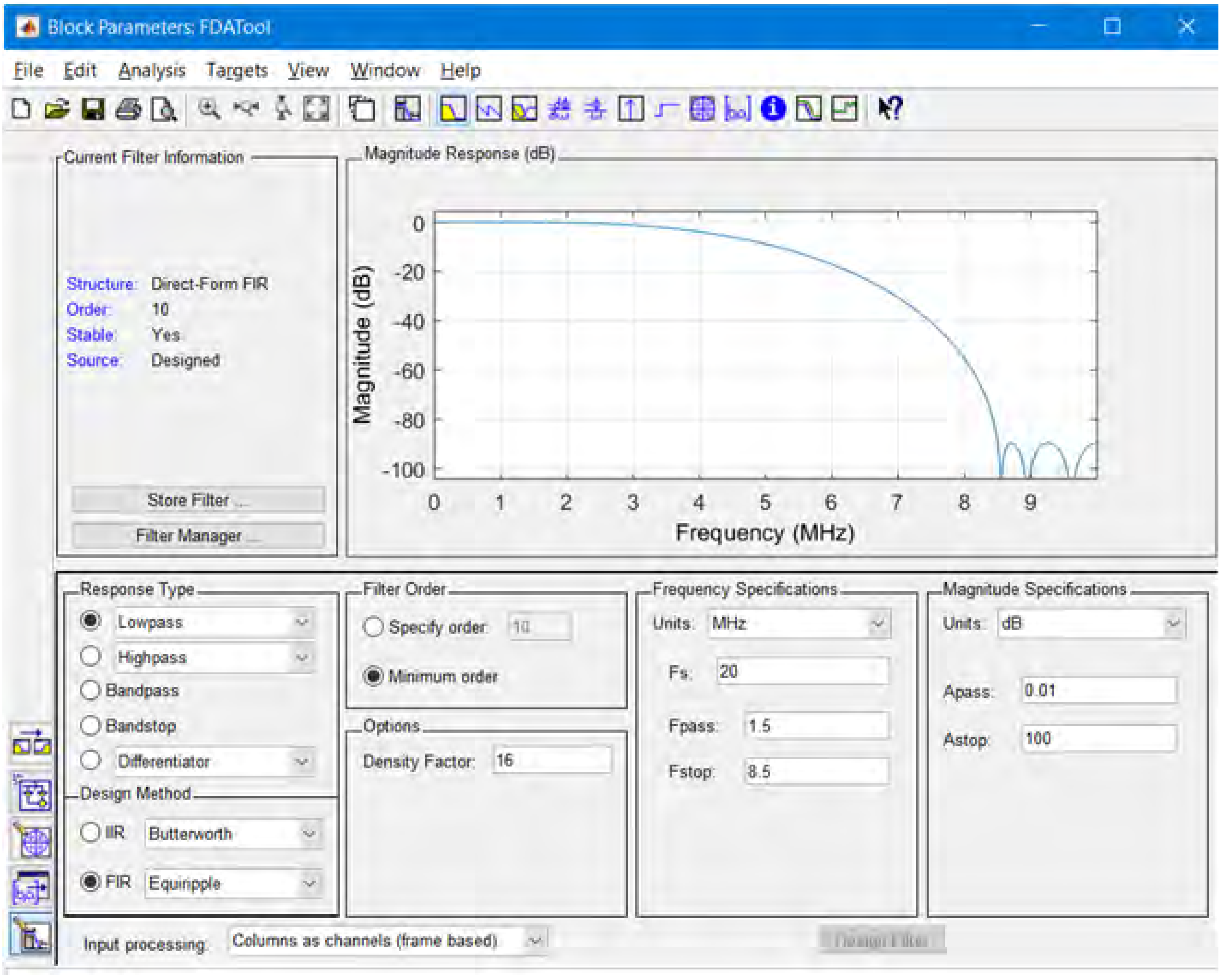
The frequency response of both filters visualized in Spectrum Analyzer block.

Figure 56: **Spectrum Analyzer Block**



The Xilinx version of the FDAtool can be used to define the coefficients of the low-pass filter to eliminate high-frequency noise. The filter configuration parameters like Response Type, Filter Order, Frequency Specification, and Magnitude Specification can be modified from the Properties Editor of the FDATool as shown below.

Figure 57: **Filter Configuration**



Structure: Direct-Form FIR
Order: 10
Stable: Yes
Source: Designed

Response Type: [✓ Lowpass]

Design Method: [✓ FIR] Equiripple

Filter Order: Specify order: 10 [✓ Minimum order]

Options: Density Factor: 16

Frequency Specifications: Units: MHz; Fs: 20; Fpass: 1.5; Fstop: 8.5

Magnitude Specifications: Units: dB; Apass: 0.01; Astop: 100

Input processing: Columns as channels (frame based)

The Design Filter option at the bottom of the tool window allows you to find out the filter order and observe the Magnitude Response. You can also view the Phase Response, Impulse Response, Coefficients, and more by selecting the appropriate icon at the top-right of the window. You can display the filter coefficients in the MATLAB® workspace by typing the following:

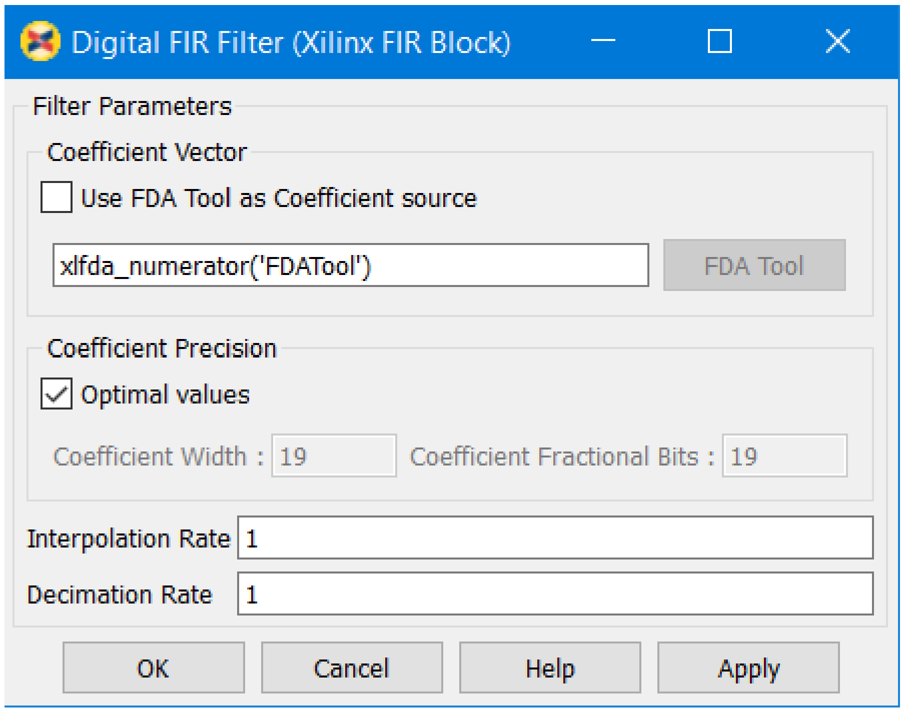
```
>> xlfda_numerator('FDATool')
```

The following functions help you find the maximum and minimum coefficient values to adequately specify the coefficient width and binary point:

```
>> max(xlfda_numerator('FDATool'))
>> min(xlfda_numerator('FDATool'))
```

Now, the filter parameters of the FDATool instance can be associated with the Digital FIR filter instance.

Figure 58: **Digital FIR Filter**



Digital FIR Filter (Xilinx FIR Block)

Filter Parameters

Coefficient Vector

☐ Use FDA Tool as Coefficient source

xlfda_numerator('FDATool')

Coefficient Precision

☑ Optimal values

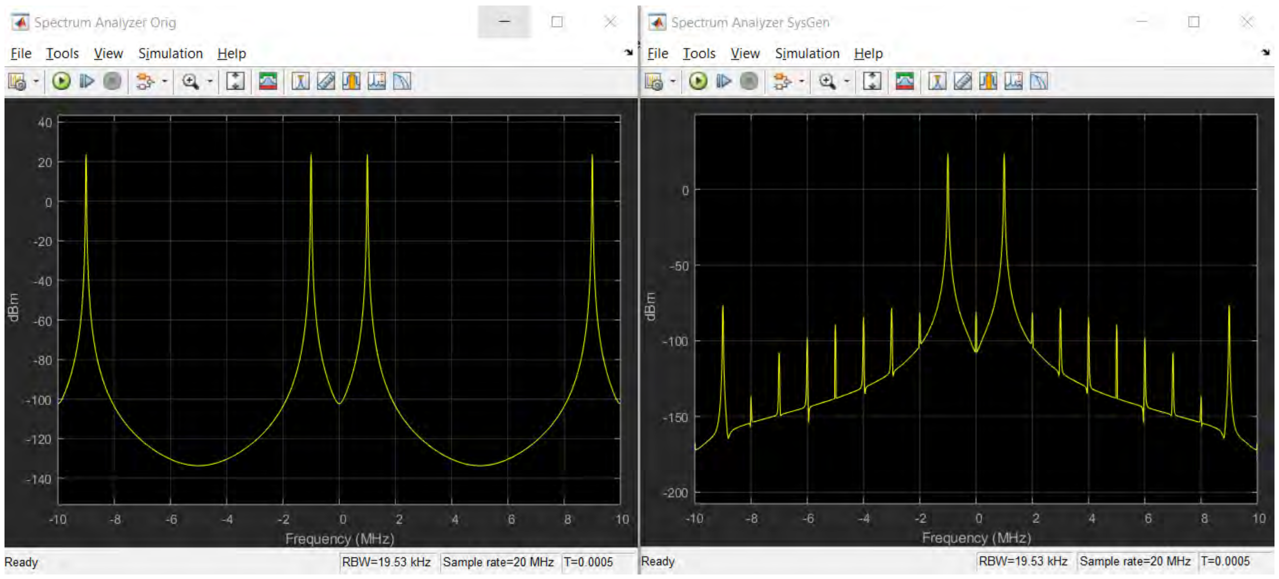
Coefficient Width : 19 Coefficient Fractional Bits : 19

Interpolation Rate: 1

Decimation Rate: 1

The Xilinx Filter response can be viewed and compared with the Simulink® response using the Spectrum Analyzer.

Figure 59: **Spectrum Analyzer**



***Note*:** The frequency response results of Model Composer (right side), shown above, differs slightly with the original design (left side) due to the quantization and sampling effect inherent when a continuous time system is described in discrete time hardware.

For complete example along with steps to use the FDATool, refer to the *Vitis Model Composer Tutorial* (UG1498).

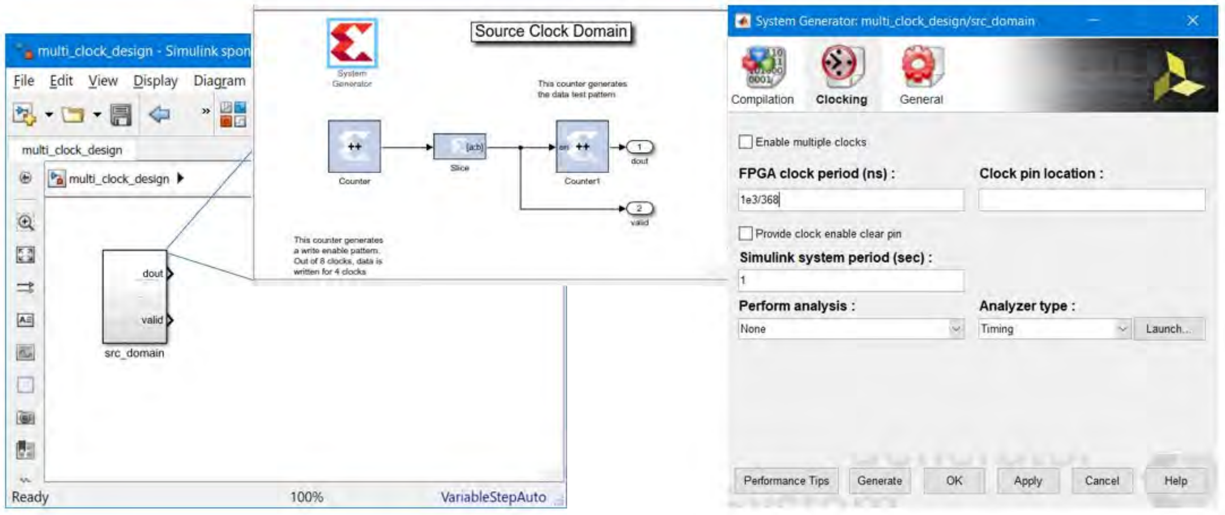
Multiple Independent Clocks Hardware Design

Model Composer is a cycle accurate, high-level hardware modeling and implementation tool where the notion of a cycle is analogous to that of clock in hardware. The design can be partitioned into groups of Subsystem blocks, where each Subsystem has a common cycle period, independent of the cycle period of other Subsystems. This section details how blocks can be grouped into one cycle or clock domain and how data can be transferred between these cycle domains. In the rest of this section, the terms cycle and clock are used interchangeably.

Grouping Blocks within a Clock Domain

Blocks are grouped together in Model Composer by using a Subsystem. Grouping blocks within a clock domain is no different except that a System Generator token has to be placed in the Subsystem you want to "mark" as a Clock Domain. This is shown in the figure below.

Figure 60: **Source Clock Domain**



Enable multiple clocks: ☐

FPGA clock period (ns) : 1e3/368

Clock pin location :

Provide clock enable clear pin: ☐

Simulink system period (sec) : 1

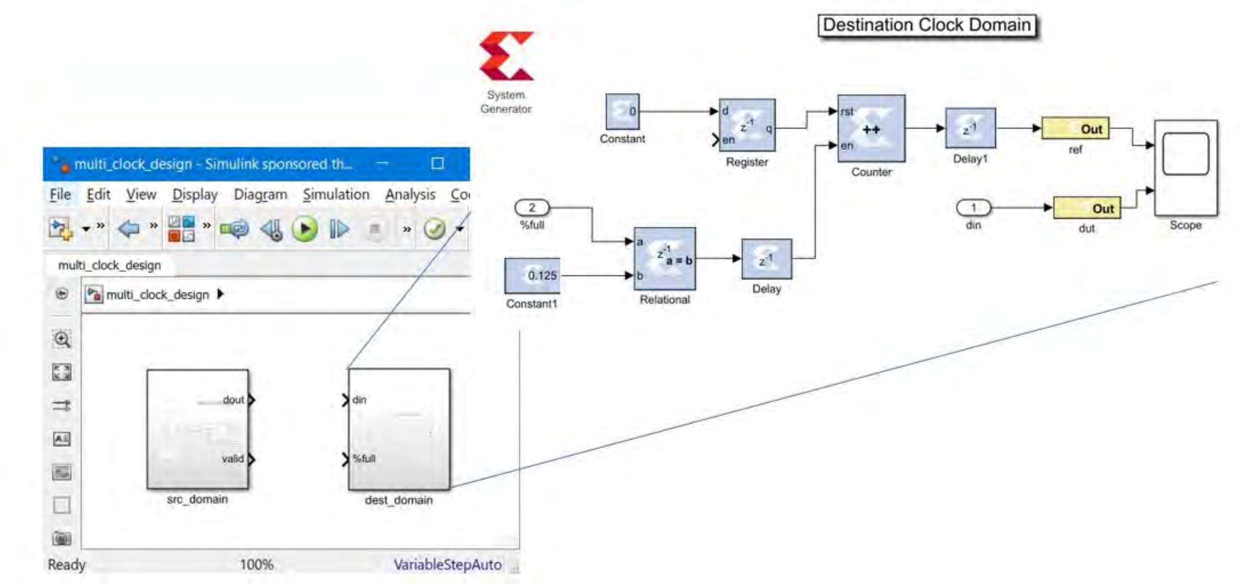
Perform analysis : None

Analyzer type : Timing

In this figure, a clock domain Subsystem called `src_domain` has been created and a System Generator token added. Notice that the clocking tab of the System Generator token is selected. In this tab, the FPGA clock period has been set to (1000/368) ns (368 MHz) and the Simulink system period to 1. This implies that an advance of 1 Simulink second corresponds to (1000/368) ns of FPGA clock.

Similarly, another group of blocks representing another clock domain is included in a Subsystem called `dest_domain`, as shown in the figure below.

Figure 61: **Destination Clock Domain**



In this design, the `dest_domain` Subsystem is configured to run at an FPGA clock period of 1000/245 ns (245 MHz). The Simulink system period is set to 368/245. This is done because the Simulink system period of the `src_domain` Subsystem is set to 1. Hence, you normalize with the System period from the `src_domain` which is faster.

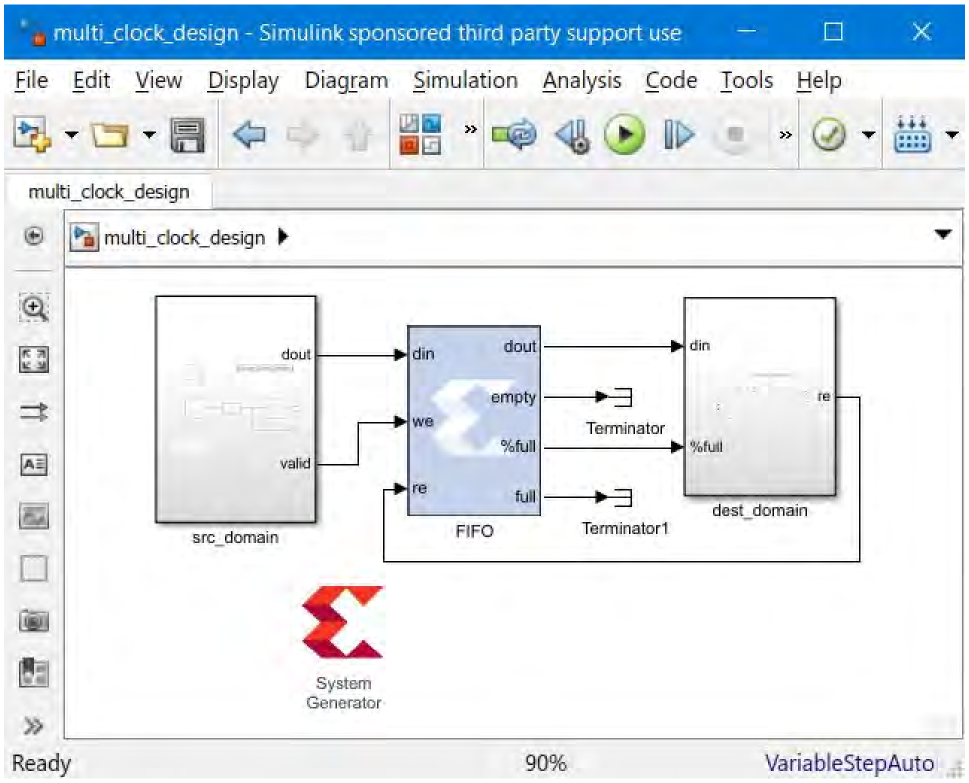
HDL Blocks used to Create Asynchronous Clock Domains

To pass data between the `src_domain` and `dest_domain` Subsystems, you can use any one of the following logics:

1. FIFO block
2. Dual Port RAM block
3. Register block
4. Black Box block, which allows existing VHDL, Verilog, and EDIF to be brought into a design. For more information about Black Box utility, please refer to Importing HDL Modules.

These blocks configure themselves to be either synchronous single clock blocks or multiple clock blocks based on their context in the design. In this design, the FIFO block is used to cross the clock domains as shown in the figure below.

Figure 62: **Cross Domain FIFO Block**

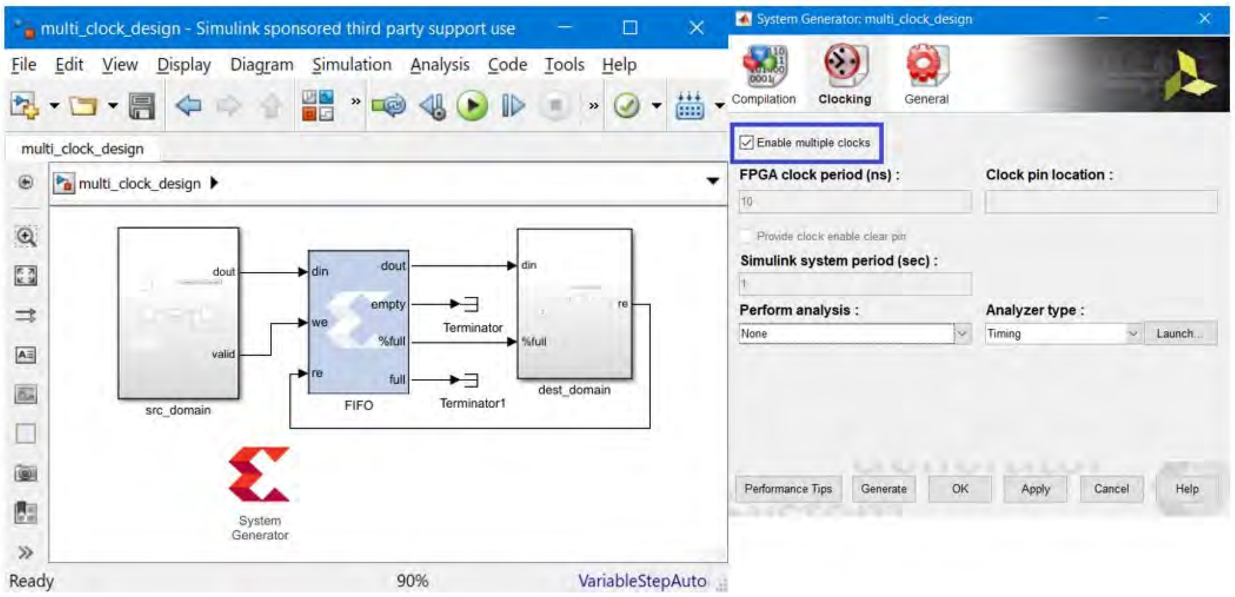


To complete the design, the FIFO block and an additional System Generator token block at the top level of the design is included to enable Code Generation.

Configuring the Top-Level System Generator Token

The top-level System Generator token has to be configured to indicate that the Code Generation must proceed for a multiple clock design. This is indicated by turning on the **Enable multiple clocks** check box in the top-level System Generator token. This indicates to the Code Generation engine that the clock information for the Subsystems `src_domain` and `dest_domain` must be obtained from the System Generator tokens contained in those Subsystems. If this check box is not enabled, then the design will be treated as a single clock design where all the clock information is inherited from the top-level System Generator token block.

Figure 63: **Enable Multiple Clocks**



☑ Enable multiple clocks
FPGA clock period (ns) : 10
Clock pin location :
Provide clock enable clear pin
Simulink system period (sec) : 1
Perform analysis : None
Analyzer type : Timing

Clock Propagation Algorithm

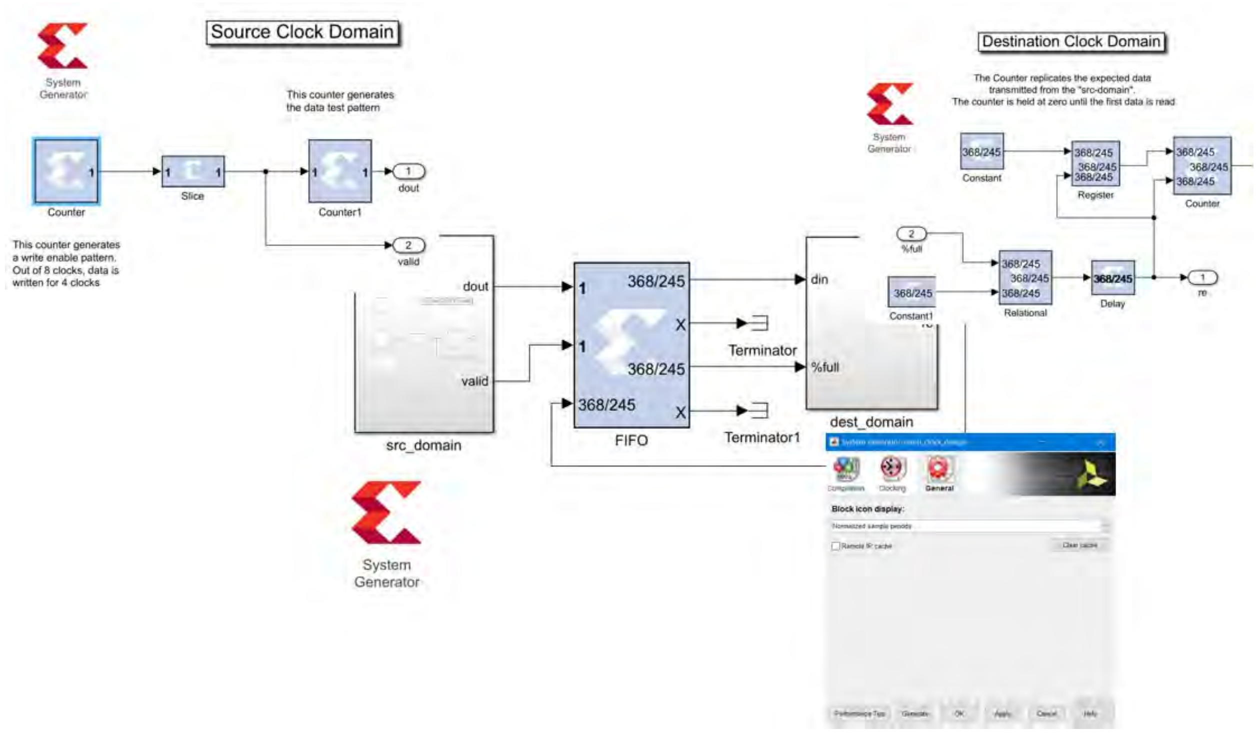
For all HDL blocks in the `src_domain`, the clocking is governed by the System Generator token in the `src_domain` Subsystem. Similarly for the `dest_domain` Subsystem. For the FIFO block, the clocks are derived from its context in the design. Because the `we` and `din` ports are driven by signals emanating from the `src_domain` Subsystem, the `wr_clk` of the FIFO is tied to the `src_domain` clock. Because the `dout`, `full`, and `re` ports either drive or load signals from `dest_domain`, the `rd_clk` of the FIFO is tied to the `dest_domain` clock. Mixing and matching these signals across clock domains or using any other block (other than FIFO or Dual Port RAM) to cross clock domains will result in a DRC error.

Debugging Clock Propagation

The top-level System Generator token can be used to control the display of all HDL Block Icons using the **Block icon display** control in the General Tab. From this tab, you can either select **Normalized sample periods** or **Sample frequencies** to help understand how clocks get propagated in the design.

For multiple clock designs, the behavior of **Normalized sample periods**, is that the smallest **Simulink system period** is used to normalize all the sample periods in the design.

Figure 64: **Debugging Clock Propagation**


Source Clock Domain
System Generator
This counter generates the data test pattern
Counter
Slice
Counter1
dout
valid
This counter generates a write enable pattern. Out of 8 clocks, data is written for 4 clocks
src_domain
368/245
FIFO
Terminator
Terminator1
din
%full
dest_domain
Destination Clock Domain
The Counter replicates the expected data transmitted from the "src-domain". The counter is held at zero until the first data is read
Constant
Register
Relational
Constant1
Delay
re
Block icon display:
Normalized sample periods
System Generator

To enable the above display, set the **Block icon display** of the top-level System Generator token to **Normalized Sample Periods** and press **Apply**.

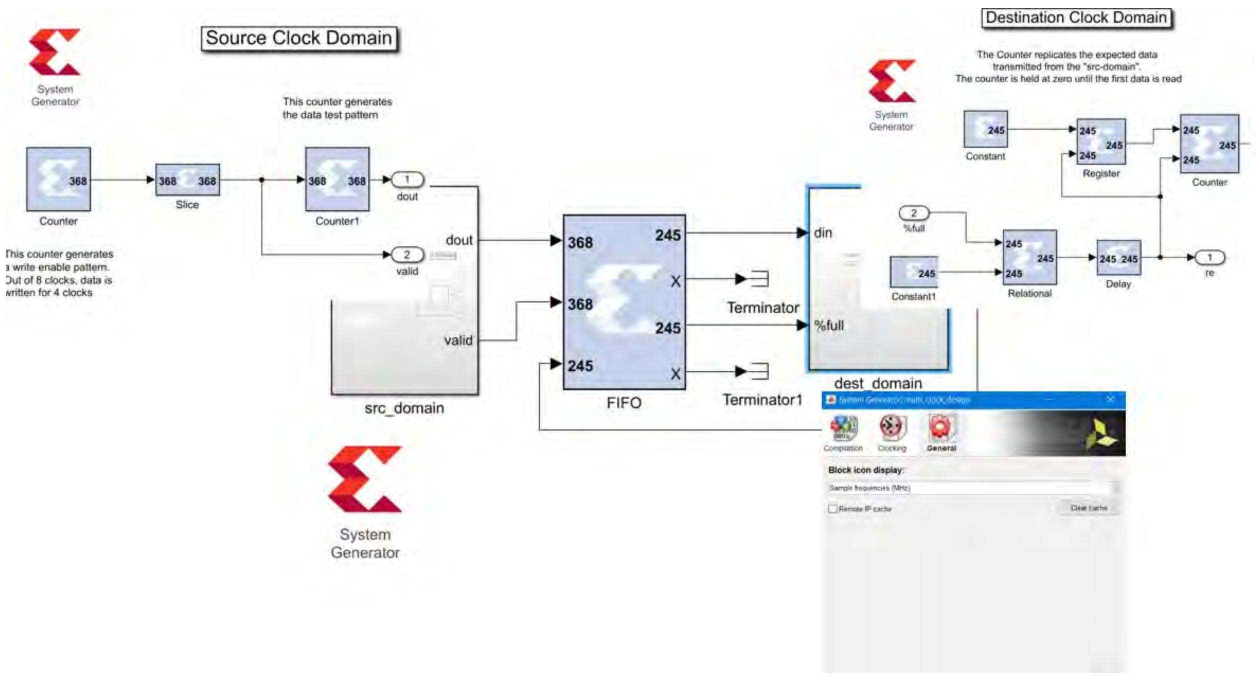
For **Sample Frequencies**, the port icon text display is the result of the following computation:

(1e6/FPGA clock period) * Simulink system period/Port sample period

where FPGA clock period is the FPGA clock period specified in ns in the domain's System Generator token, and Simulink system period is the Simulink system period in seconds specified in the domain's System Generator token.

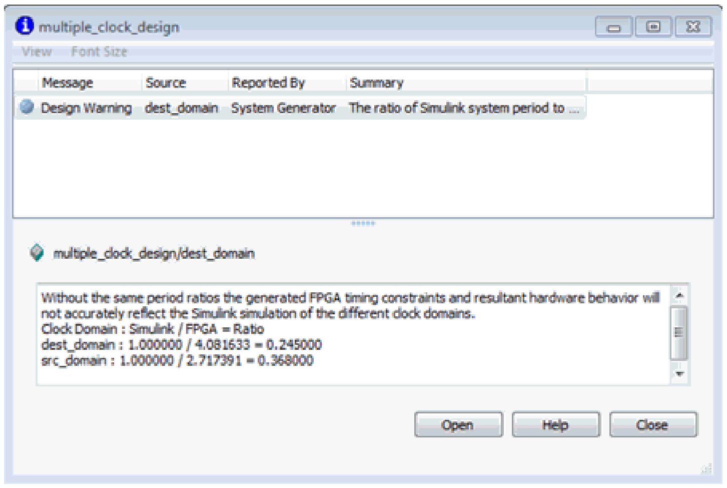
The **Sample Frequencies** can also be used to validate correct clock propagation as shown in the following figure:

Figure 65: **Sample Frequencies**



To ensure that the simulation models the hardware behavior relatively with respect to the clocks, the ratio of Simulink system period to FPGA clock period in each domain must be the same. If this relationship is not complied with the correct ratio, a warning is thrown to indicate this problem as shown in the figure below:

Figure 66: **Warning**



Message	Source	Reported By	Summary
Design Warning	dest_domain	System Generator	The ratio of Simulink system period to ...

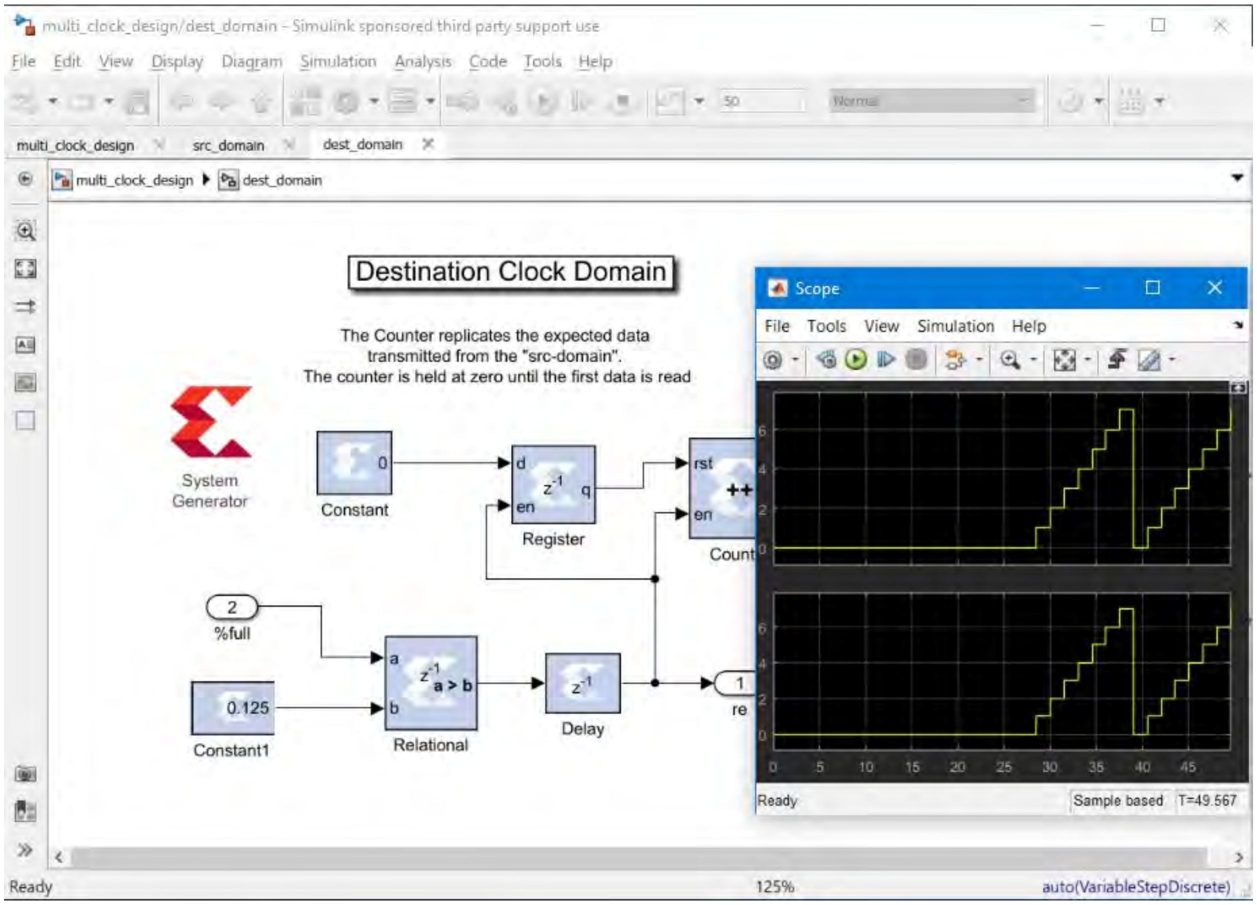
multiple_clock_design/dest_domain

```
Without the same period ratios the generated FPGA timing constraints and resultant hardware behavior will
not accurately reflect the Simulink simulation of the different clock domains.
Clock Domain : Simulink / FPGA = Ratio
dest_domain : 1.000000 / 4.081633 = 0.245000
src_domain : 1.000000 / 2.717391 = 0.368000
```

Simulation

After performing the simulation, the following results are obtained as seen in the `dest_domain` scope.

Figure 67: **Simulation**


multi_clock_design/dest_domain - Simulink sponsored third party support use
File Edit View Display Diagram Simulation Analysis Code Tools Help
multi_clock_design
src_domain
dest_domain
Destination Clock Domain
The Counter replicates the expected data transmitted from the "src-domain". The counter is held at zero until the first data is read
System Generator
Constant
Register
Count
%full
Constant1
Relational
Delay
re
Scope
File Tools View Simulation Help
Ready
Sample based
T=49.567
Ready
125%
auto(VariableStepDiscrete)

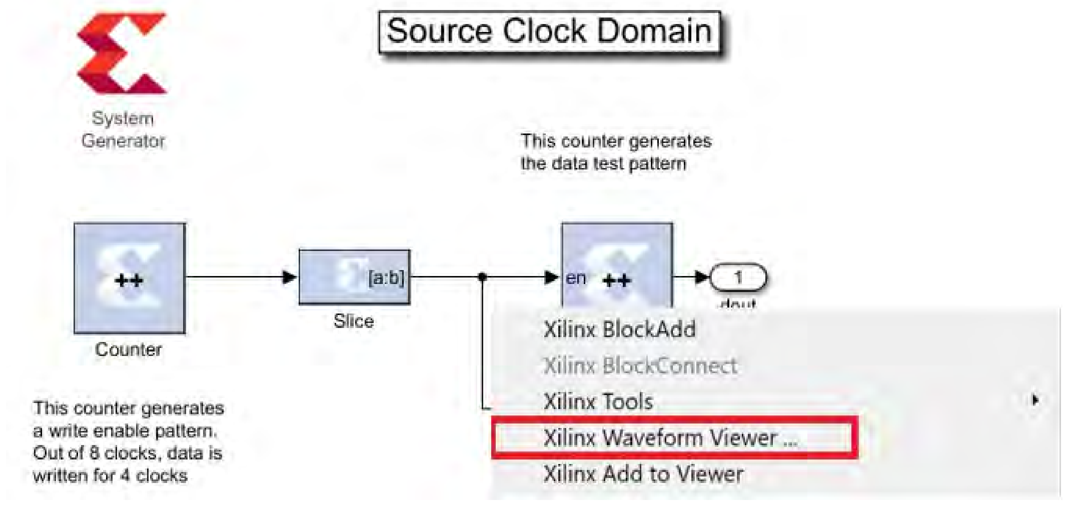
As shown above, the simulation results indicate that the data obtained is the data expected.

Note: This cross-clock domain simulation behavior is NOT cycle accurate.

Debugging Multiple Clock Domain Signals

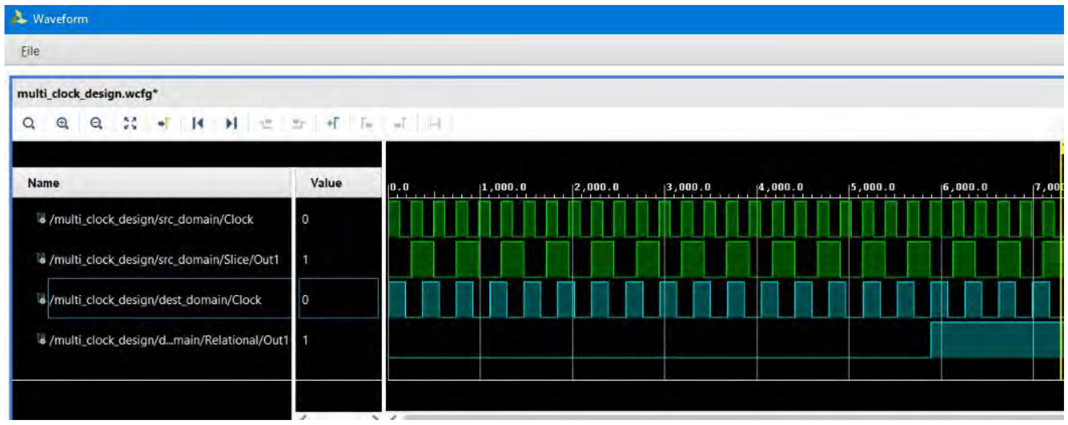
In Model Composer you can use cross probing between the signal in the Xilinx Waveform Viewer and the Simulink® diagram to aid the debugging process.

Figure 68: **Source Clock Domain**


Source Clock Domain
System Generator
This counter generates the data test pattern
Counter
Slice
[a:b]
en
1
dout
This counter generates a write enable pattern. Out of 8 clocks, data is written for 4 clocks
Xilinx BlockAdd
Xilinx BlockConnect
Xilinx Tools
Xilinx Waveform Viewer ...
Xilinx Add to Viewer

To add a signal to the Waveform viewer, right-click the signal in the model and select **Xilinx Add To Viewer**. Simulating the design should launch the Waveform Viewer as shown below.

Figure 69: **Waveform Viewer**



Name	Value
/multi_clock_design/src_domain/Clock	0
/multi_clock_design/src_domain/Slice/Out1	1
/multi_clock_design/dest_domain/Clock	0
/multi_clock_design/d...main/Relational/Out1	1

All signals in same clock domain are colored similarly. In the figure above: `src_domain/Slice/Out1` and `dest_domain/Relational/Out1` are in different clock domains.

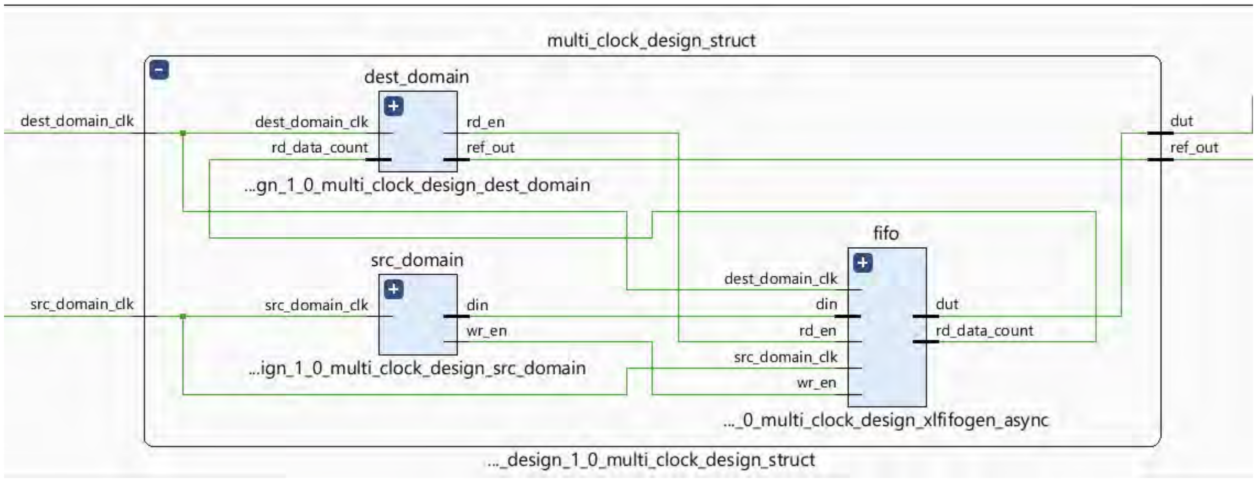
Code Generation

Code generation for a Multiple Clock design supports the following compilation targets:

- **HDL Netlist**
- **IP Catalog**
- **Synthesized Checkpoint**

A screen shot of the top-level hardware is shown in the figure below.

Figure 70: **Top-Level Hardware**


multi_clock_design_struct
dest_domain
dest_domain_clk
dest_domain_clk
rd_data_count
rd_en
ref_out
...gn_1_0_multi_clock_design_dest_domain
dut
ref_out
fifo
src_domain
src_domain_clk
src_domain_clk
din
wr_en
...ign_1_0_multi_clock_design_src_domain
dest_domain_clk
din
rd_en
src_domain_clk
wr_en
dut
rd_data_count
..._0_multi_clock_design_xlfifogen_async
..._design_1_0_multi_clock_design_struct

As many clock ports as there are clock domains are exposed at the top level and can be driven by a variety of Xilinx clocking constructs like MMCM, PLL etc. It is assumed that these clocks are completely asynchronous and the following period constraints are created:

```
create_clock -name src_domain_clk -period 2.717 [get_ports src_domain_clk]
create_clock -name dest_domain_clk -period 4.082 [get_ports dest_domain_clk]
```

These are the only constraints that are required because only FIFO or Dual Port RAM are allowed which have any additional clock domain constraints embedded in the IP.

Known Issues

The following are some of the known issues:

- The HWCosim Compilation Target is not supported for Multiple Clock Designs.
- Only FIFO & Dual Port RAM blocks can be in the top-level of the design when using multiple clocks.
- The behavior of blocks that aid in the crossing of Multiple clock domains is NOT cycle accurate.
- Unconnected or terminated output ports cannot be viewed in the Waveform Viewer.

AXI Interface

AMBA AXI4 (Advanced eXtensible Interface 4) is the fourth generation of the AMBA interface defined and controlled by Arm®, and has been adopted by Xilinx as the next-generation interconnect for FPGA designs. Xilinx and Arm worked closely to ensure that the AXI4 specification addresses the needs of FPGAs.

AXI is an open interface standard that is widely used by many 3rd-party IP vendors since it is public, royalty-free and an industry standard.

The AMBA AXI4 interface connections are point-to-point and come in three different flavors: AXI4, AXI4-Lite Slave, and AXI4-Stream.

- AXI4 is a memory-mapped interface which support burst transactions
- AXI4-Lite Slave is a lightweight version of AXI4 and has a non-bursting interface
- AXI4-Stream is a high-performance streaming interface for unidirectional data transfers (from master to slave) with reduced signaling requirements (compared to AXI4). AXI4-Stream supports multiple channels of data on the same set of wires.

In the following documentation, AXI4 refers to the AXI4 memory map interface, and AXI4-Lite Slave and AXI4-Stream each refer to their respective flavor of the AMBA AXI4 interface. When referring to the collection of interfaces, the term AMBA AXI4 shall be used.

The purpose of this section is to provide an introduction to AMBA AXI4 and to draw attention to AMBA AXI4 details with respect to Model Composer. For more detailed information on the AMBA AXI4 specification please refer to the Xilinx AMBA-AXI4 documents found on the AMBA AXI4 Interface Protocol page on the Xilinx website.

AXI4-Stream Support in Model Composer

The three most common AXI4-Stream signals are TVALID, TREADY and TDATA. Of all the AXI4-Stream signals, only TVALID is denoted as mandatory, all other signals are optional. All information-carrying signals propagate in the same direction as TVALID; only TREADY propagates in the opposite direction.

Since AXI4-Stream is a point-to-point interface, the concept of master and slave interface is pertinent to describe the direction of data flow. A master produces data and a slave consumes data.

Naming Conventions

AXI4-Stream signals are named in the following manner:

```
<Role>_<ClassName>[_<BusName>]_[<ChannelName>]<SignalName>
```

For example:

```
m_axis_tvalid
```

Here `m` denotes the Role (master), `axis` the ClassName (AXI4-Stream) and `tvalid` the SignalName.

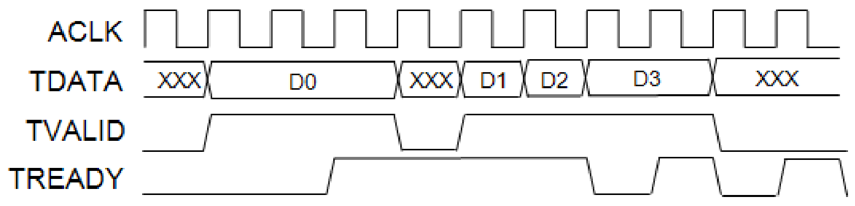
```
s_axis_control_tdata
```

Here `s` denotes the Role (slave), `axis` the ClassName, `control` the BusName which distinguishes between multiple instances of the same class on a particular IP, and `tdata` the SignalName.

Notes on TREADY/TVALID Handshaking

The TREADY/TVALID handshake is a fundamental concept in AXI to control how data is exchanged between the master and slave allowing for bidirectional flow control. TDATA, and all the other AXI4-Stream signals (TSTRB, TUSER, TLAST, TID, and TDEST) are all qualified by the TREADY/TVALID handshake. The master indicates a valid beat of data by the assertion of TVALID and must hold the data beat until TREADY is asserted. TVALID once asserted cannot be de-asserted until TREADY is asserted in response (this behavior is referred to as a "sticky" TVALID). AXI also adds the rule that TREADY can depend on TVALID, but the assertion of TVALID cannot depend on TREADY. This rule prevents circular timing loops. The timing diagram below provides an example of the TREADY/TVALID handshake.

Figure 71: **TREADY/TVALID Handshake Timing Diagram**



Handshaking Key Points

- A transfer on any given channel occurs when both TREADY and TVALID are high in the same cycle.
- TVALID once asserted, may only be de-asserted after a transfer has completed (TREADY is sampled high). Transfers may not be retracted or aborted.
- Once TVALID is asserted, no other signals in the same channel (except TREADY) may change value until the transfer completes (the cycle after TREADY is asserted).
- TREADY may be asserted before, during or after the cycle in which TVALID is asserted.
- The assertion of TVALID may not be dependent on the value of TREADY. But the assertion of TREADY may be dependent on the value of TVALID.

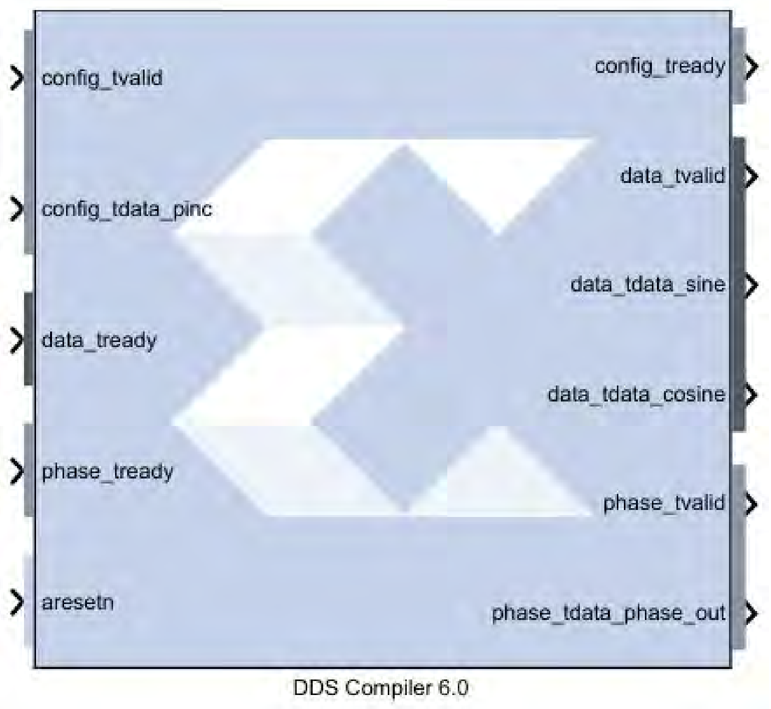
- There must be no combinatorial paths between input and output signals on both master and slave interfaces:
    - Applied to AXI4-Stream IP, this means that the TREADY slave output cannot be combinatorially generated from the TVALID slave input. A slave that can immediately accept data qualified by TVALID, should pre-assert its TREADY signal until data is received. Alternatively TREADY can be registered and driven the cycle following TVALID assertion.
    - The default design convention is that a slave should drive TREADY independently or pre-assert TREADY to minimize latency.
    - Note that combinatorial paths between input and output signals are permitted across separate AXI4-Stream channels. It is however a recommendation that multiple channels belonging to the same interface (related group of channels that operate together) should not have any combinatorial paths between input and output signals.
- For any given channel, all signals propagate from the source (typically master) to the destination (typically slave) except for TREADY. Any other information-carrying or control signals that need to propagate in the opposite direction must either be part of a separate channel ("back-channel" with separate TREADY/TVALID handshake) or be an out-of-band signal (no handshake). TREADY should not be used as a mechanism to transfer opposite direction information from a slave to a master.
- AXI4-Stream allows TREADY to be omitted which defaults its value to 1. This may limit interoperability with IP that generates TREADY. It is possible to connect an AXI4-Stream master with only forward flow control (TVALID only).

AXI4-Stream Blocks in Model Composer

HDL blocks that present an AXI4-Stream interface can be found in the Vitis Model Composer HDL Blockset Library entitled DSP/AXI4. Blocks in this library are drawn slightly differently from regular (non AXI4-Stream) blocks.

Port Groupings

Figure 72: **DDS Compiler 6.0**



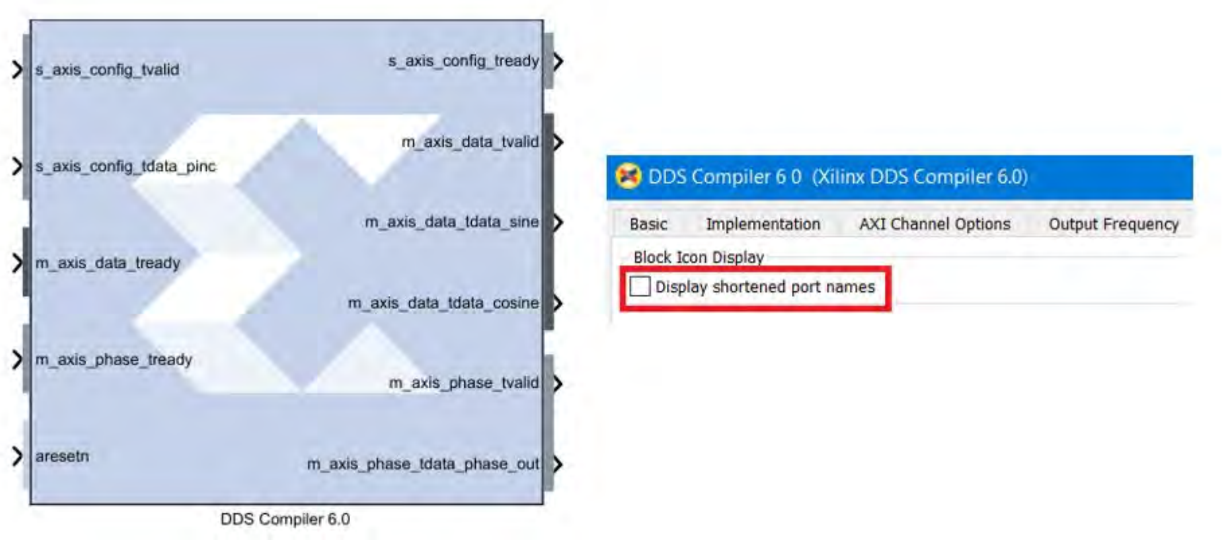
Blocks that offer AXI4-Stream interfaces have AXI4-Stream channels grouped together and color coded. For example, on the DDS compiler 6.0 block shown above, the input port `data_tready`, and the three output ports, `data_tvalid`, `data_tdata_sine`, `data_tdata_cosine` belong in the same AXI4-Stream channel. Similarly, the input port `config_tvalid`, `config_tdata_pinc` and output port `config_tready` belong in the same AXI4-Stream channel. As does `phase_tready`, `phase_tvalid`, and `phase_tdata_phase_out`.

Signals that are not part of any AXI4-Stream channels are given the same background color as the block; `aresetn` is an example.

Port Name Shortening

In the example shown below, the AXI4-Stream signal names have been shortened to improve readability on the block. Name shortening is purely cosmetic and when netlisting occurs, the full AXI4-Stream name is used. Name shorting is turned on by default; you can uncheck the **Display shortened port names** option in the block parameter dialog box to reveal the full name.

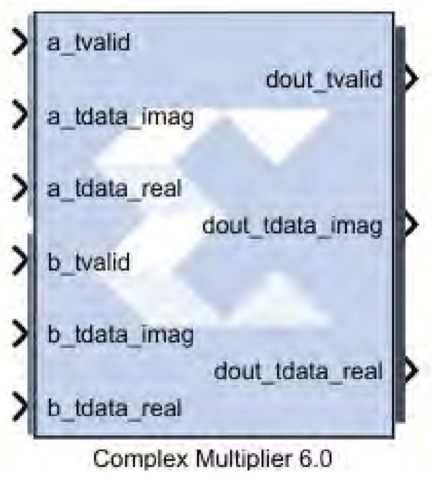
Figure 73: **DDS Compiler 6.0**



Breaking Out Multi-Channel TDATA

In AXI4-Stream, TDATA can contain multiple channels of data. In Model Composer, the individual channels for TDATA are broken out. So for example, the TDATA of port `dout` below contains both real and imaginary components.

Figure 74: **Complex Multiplier 6.0**



The breaking out of multi-channel TDATA does not add additional logic to the design and is done in Model Composer as a convenience to the users. The data in each broken out TDATA port is also correctly byte-aligned.

AXI4-Lite Slave Interface Generation

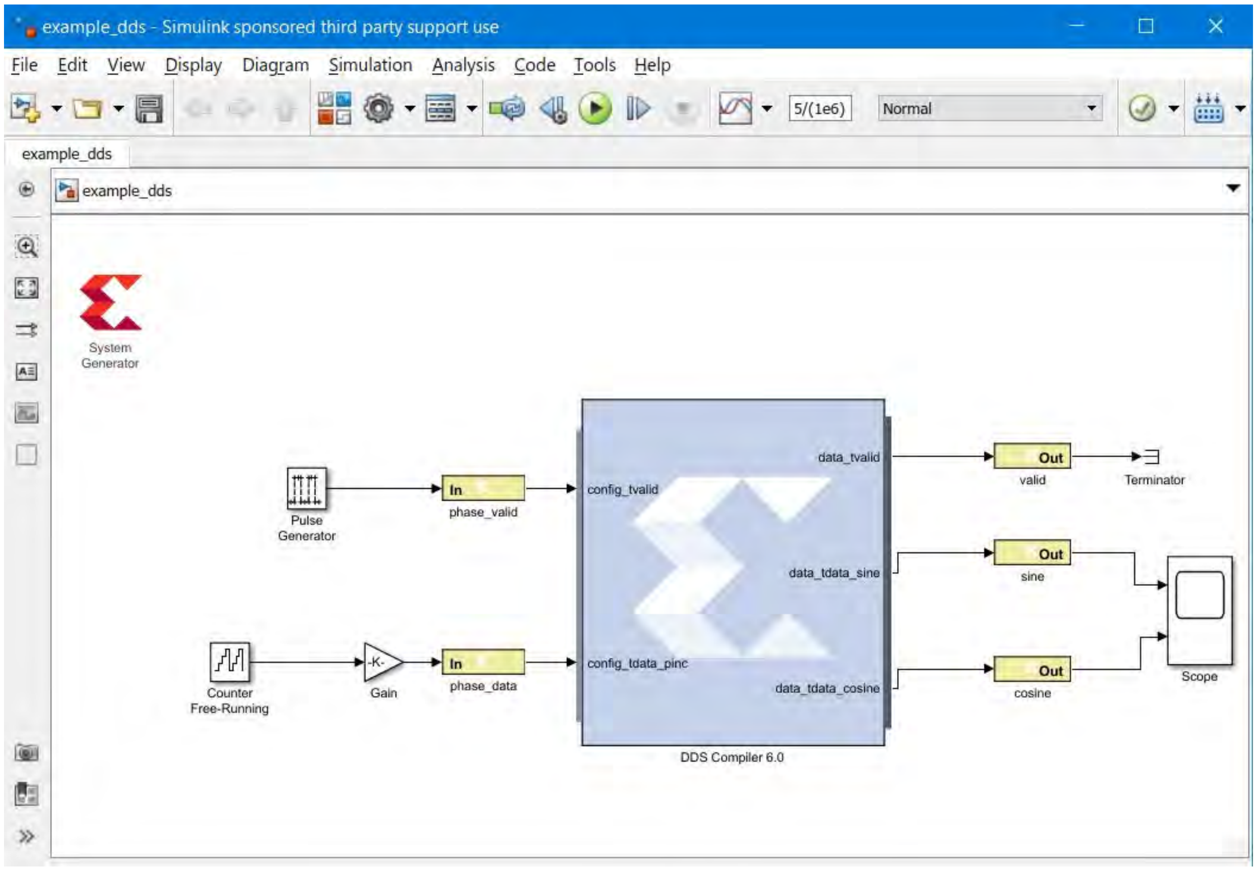
Design modules that are developed using Model Composer usually form a Subsystem of a larger DSP or Video system. These Model Composer modules are typically algorithmic and data path heavy modules that are best created in the visually-rich environment like MATLAB/Simulink. The larger system is typically assembled from IP from the Vivado® IP catalog. These IP typically use standard stream and control interfaces like AXI4-Lite Slave and the larger system is typically assembled using a tool like the Vivado IP integrator.

This topic describes features in Model Composer that allow you to create a standard AXI4-Lite Slave interface for a Model Composer module and then export the module to the Vivado® IP catalog for later inclusion in a larger design using IP integrator. Model Composer also allows creation of multiple AXI4-Lite Slave interfaces across multiple clock domains.

AXI4-Lite Interface Synthesis in Model Composer

Design creation and verification is exactly the same as any other Model Composer design that does not include an AXI4-Lite interface. Consider the `example_dds` design shown below.

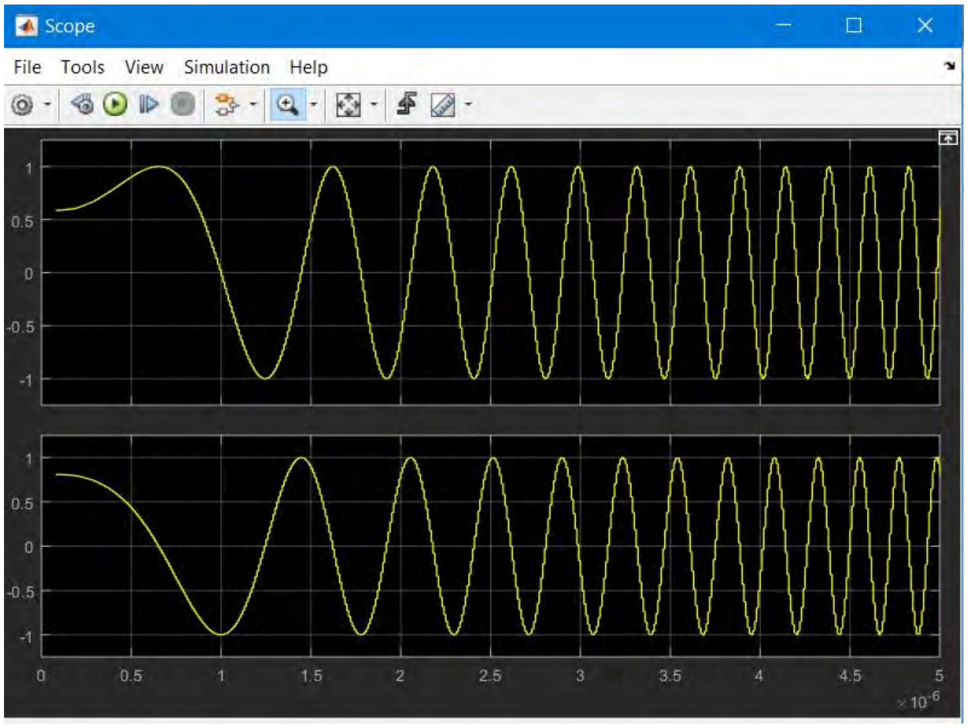
Figure 75: **Example DDS Design**



This design contains a DDS Compiler where the two input ports, `config_tvalid` and `config_tdata_pinc` are used to control the output frequency.

Following are the simulation results of this design which indicate both sine and cosine output separately.

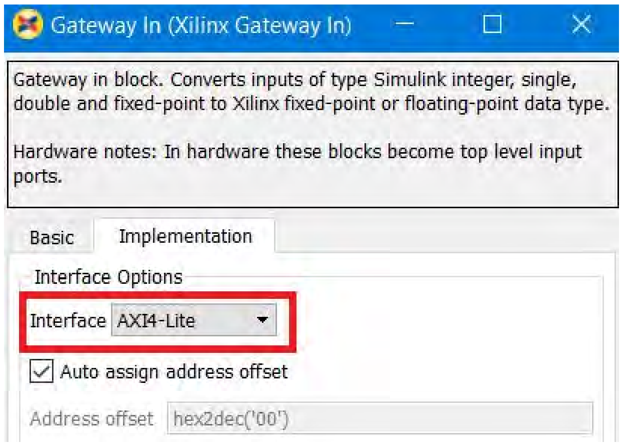
Figure 76: **Simulation Results**



Configuring the Design for an AXI4-Lite Interface

In the example_dds design, Gateway In and Gateway Out blocks mark the boundary of the Cycle and Bit accurate FPGA portion of the Simulink design. Control of the DDS Compiler frequency is accomplished by “injecting” the correct value on the signals attached to the output port of Gateway In’s called `phase_valid` and `phase_data`. This is accomplished by modifying the Interface Options, as shown below for the `phase_valid` block.

Figure 77: **Interface Options**



Gateway In (Xilinx Gateway In)

Gateway in block. Converts inputs of type Simulink integer, single, double and fixed-point to Xilinx fixed-point or floating-point data type.

Hardware notes: In hardware these blocks become top level input ports.

Interface: AXI4-Lite

[✓] Auto assign address offset

Address offset: hex2dec('00')

As you can see, the Interface is specified as a slave AXI4-Lite Interface in Model Composer, which means that it will be transformed to a top-level AXI4-Lite interface.

The following options are also of particular interest:

Auto assign address offset (Enabled): Each Gateway is associated with a register within the AXI4-Lite Interface and this control specifies that Automatic assignment of address offsets will take place in the design based on number of different Gateway Ins mapped to the AXI4-Lite interface. Addresses are byte aligned to a 32-bit data width.

Address offset (Disabled): This option is only enabled if **Auto assign address offset** is unchecked. It allows the user to manually override of Address Offset.

Interface Name: Assigns a unique name to this interface. This name can be used to differentiate between multiple AXI4-Lite interfaces in the design.

IMPORTANT! *The Interface Name must be composed of alphanumeric characters (lowercase alphabetic) or an underscore (_) only, and must begin with a lowercase alphabetic character. axi4_lite1 is acceptable, 1Axi4-Lite is not.*

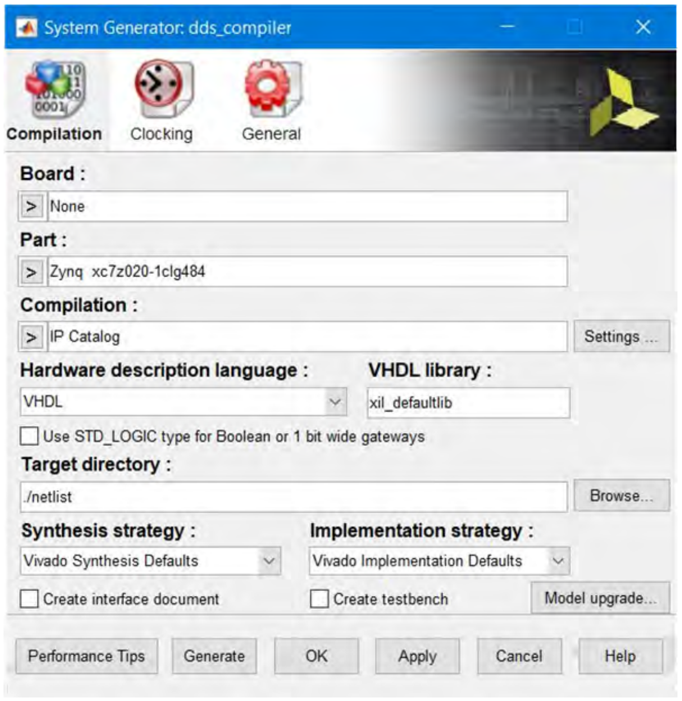
Description: The text you enter here is captured in the "Interface Documentation" that is automatically created when the design is exported to the Vivado IP catalog.

Configure the other Gateways in the design in a similar fashion.

Packaging the Design for Use in Vivado IP Integrator

When you complete the verification in Model Composer, you can package the design for use in IP integrator.

Figure 78: **Model Composer Verification**

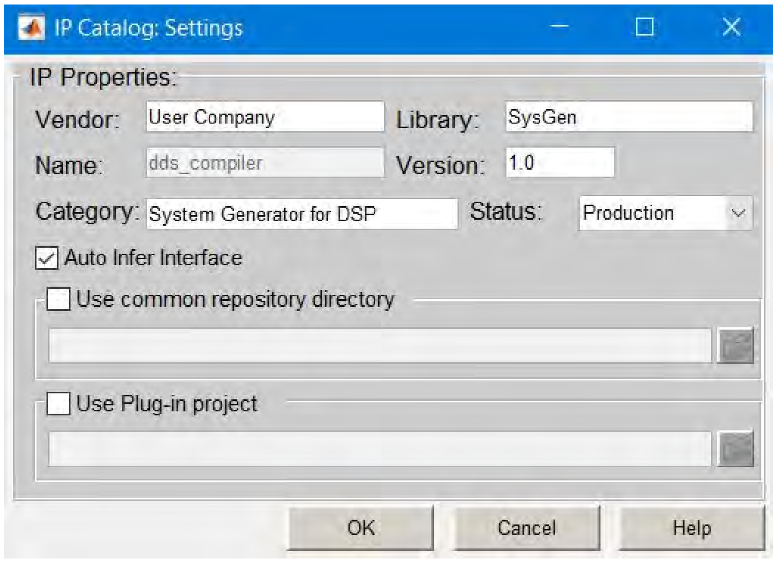


Board :
None
Part :
Zynq xc7z020-1clg484
Compilation :
IP Catalog
Hardware description language :
VHDL
VHDL library :
xil_defaultlib
Use STD_LOGIC type for Boolean or 1 bit wide gateways [ ]
Target directory :
./netlist
Synthesis strategy :
Vivado Synthesis Defaults
Implementation strategy :
Vivado Implementation Defaults
Create interface document [ ]
Create testbench [ ]

The HDL block must first be configured to a Compilation target of **IP Catalog** (the default generation target). This compilation target will consolidate all hardware source created from Model Composer (RTL + IP + Constraints) into an IP.

The part selected is the same part as that available on the Xilinx Zynq-7000 ZC702 Evaluation Board. In addition, you may also use the **Settings** button on the System Generator token to change the information that goes along with the IP. In this case, the default values shown below are used.

Figure 79: **IP Catalog Settings**



IP Properties:

Vendor:	User Company	Library:	SysGen
Name:	dds_compiler	Version:	1.0
Category:	System Generator for DSP	Status:	Production

☑ Auto Infer Interface

☐ Use common repository directory

☐ Use Plug-in project

When you click the **Generate** button in System Generator token GUI, RTL+IP+Constraints generation, as well as packaging takes place.

Description of the Generated Results

Based on the Model Composer settings shown above, the following folders and files are created.

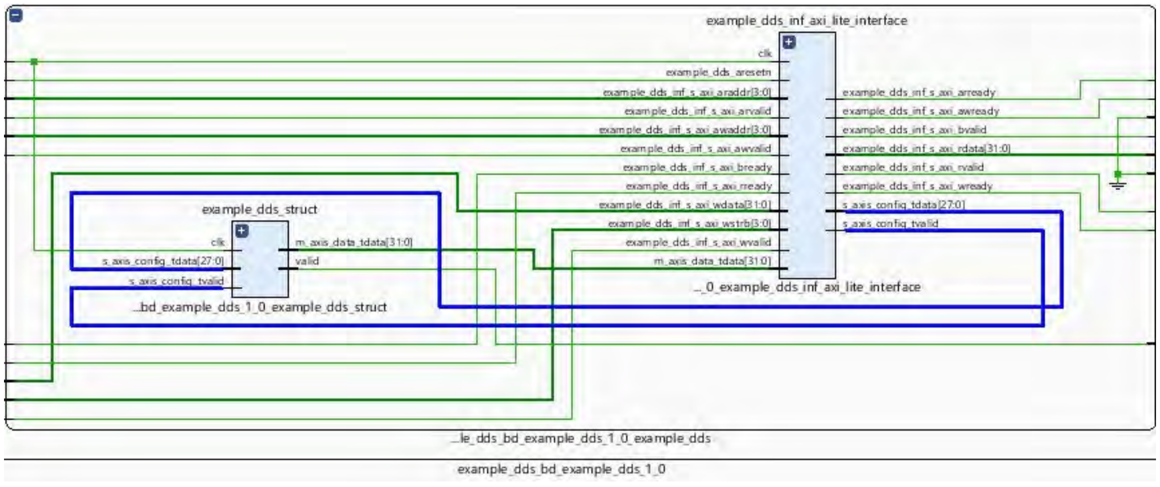
- `<target directory>/ip`: This directory contains all the IP-related hardware files, as well as the software drivers. It is this directory that you must add to the IP Catalog.
- `<target directory>/ip_catalog`: This directory contains an example Vivado IDE project called `example_dds.xpr`.

Mapping to AXI4-Lite Interfaces

Gateway Ins and Gateway Outs that are tagged as AXI4-Lite registers are mapped to different 32-bit registers within a Memory Map as shown in the Schematic below.

The schematic below is an example of mapping to a single AXI4-Lite interface, assuming all gateways have the same interface name. In a schematic with multiple AXI4-Lite interfaces, for each group of gateways having the same interface name you would see a separate AXI4-Lite Interface.

Figure 80: **Single AXI4-Lite Interface**


example_dds_inf_axi_lite_interface
clk
example_dds_aresetn
example_dds_inf_s_axi_araddr[3:0]
example_dds_inf_s_axi_arvalid
example_dds_inf_s_axi_awaddr[3:0]
example_dds_inf_s_axi_awvalid
example_dds_inf_s_axi_bready
example_dds_inf_s_axi_rready
example_dds_inf_s_axi_wdata[31:0]
example_dds_inf_s_axi_wstrb[3:0]
example_dds_inf_s_axi_wvalid
m_axis_data_tdata[31:0]
example_dds_inf_s_axi_arready
example_dds_inf_s_axi_awready
example_dds_inf_s_axi_bvalid
example_dds_inf_s_axi_rdata[31:0]
example_dds_inf_s_axi_rvalid
example_dds_inf_s_axi_wready
s_axis_config_tdata[27:0]
s_axis_config_tvalid
..._0_example_dds_inf_axi_lite_interface
example_dds_struct
clk
s_axis_config_tdata[27:0]
s_axis_config_tvalid
m_axis_data_tdata[31:0]
valid
...bd_example_dds_1_0_example_dds_struct
...le_dds_bd_example_dds_1_0_example_dds
example_dds_bd_example_dds_1_0

As you can see in the diagram, a module called `example_dds_inf_axi_lite_interface` is inserted into the design RTL, and drives the `config_tvalid` and `config_tdata` ports of the Model Composer design. And at the top level, a slave AXI4-Lite Interface is exposed. It is within this module that address decoding is done and the `config_tvalid` and `config_tdata` ports are driven based on the address obtained from the processor.

The number of bits required for addressing (`s_axi_araddr` and `s_axi_awaddr`) is determined by the number of AXI4-Lite interface registers and the offset specifications of each AXI4-Lite register. Enough bits are provided during module generation to uniquely decode each register. In this example, there are two Gateways – `phase_data` and `phase_valid`. Each port is assigned an address offset of 0x0000 & 0x0004. Hence three address bits are allocated.

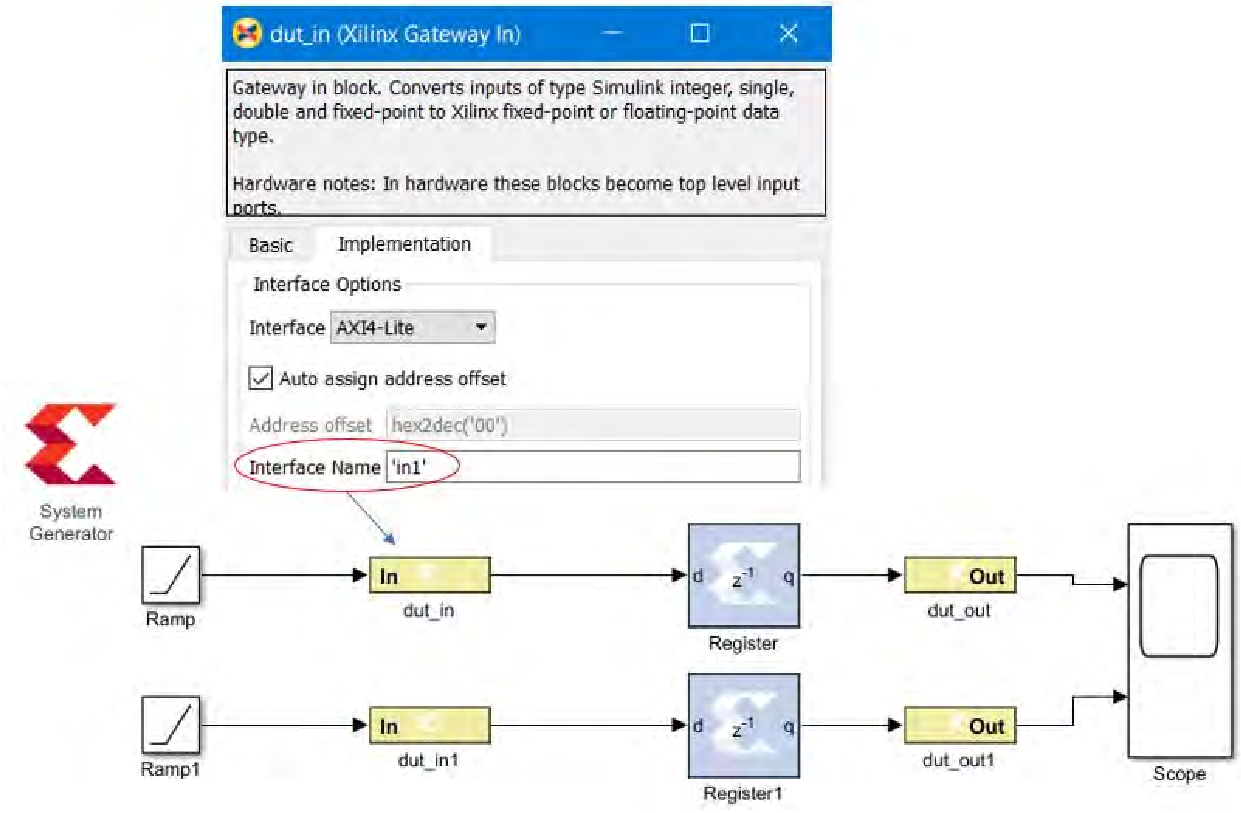
Managing Multiple AXI4-Lite Interfaces

Model Composer supports creation of IP with multiple AXI4-Lite interfaces. You can group Gateway In and Gateway Out blocks into different AXI4-Lite interfaces. This feature can be used in Multiple Clock designs as well. Software drivers will also be provided.

To assign a name to an AXI4-Lite interface, use the **Interface Name** control for the Gateway In and Gateway Out blocks associated with the interface.

All Gateway Ins and Gateway Outs with the same **Interface Name** are grouped into one AXI4-Lite Interface. An **Interface Name** must begin with a lower case alphabetic character, and can only contain alphanumeric characters (lowercase alphabetic) or an underscore (_). Having the same **Interface Name** across multiple clock domains is not supported.

Figure 81: **Interface Name**


dut_in (Xilinx Gateway In)

Gateway in block. Converts inputs of type Simulink integer, single, double and fixed-point to Xilinx fixed-point or floating-point data type.

Hardware notes: In hardware these blocks become top level input ports.

Basic | Implementation

Interface Options

Interface AXI4-Lite

☑ Auto assign address offset

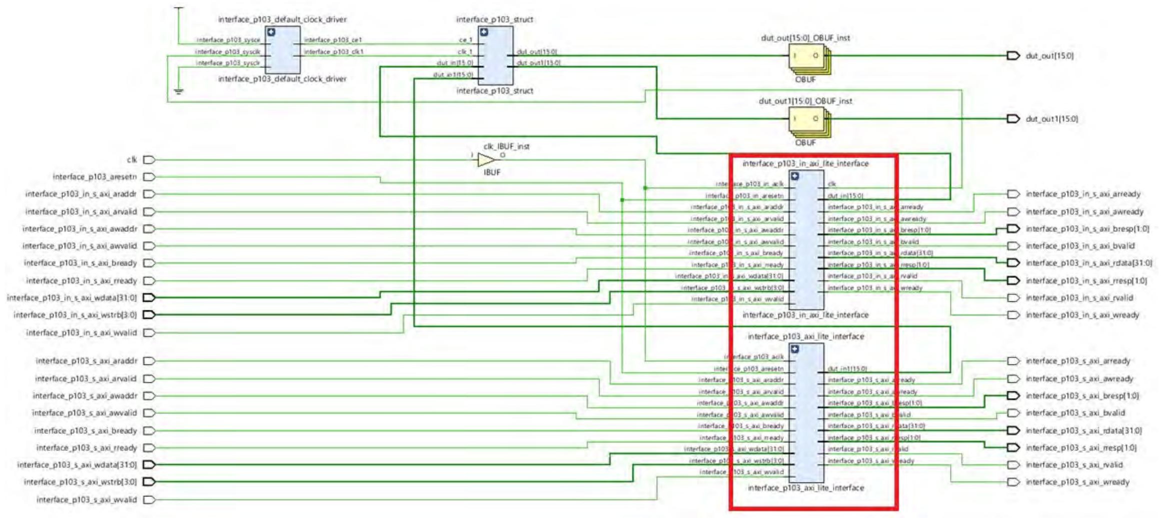
Address offset hex2dec('00')

Interface Name 'in1'

To generate the netlist you can use the **IP Catalog** or the **HDL Netlist** compilation target.

If you specify the **HDL Netlist** compilation target in the System Generator token, and then elaborate the design in Vivado, two AXI4-Lite Decoders will be created, as shown in red rectangle in the following figure.

Figure 82: **AXI4-Lite Decoders**

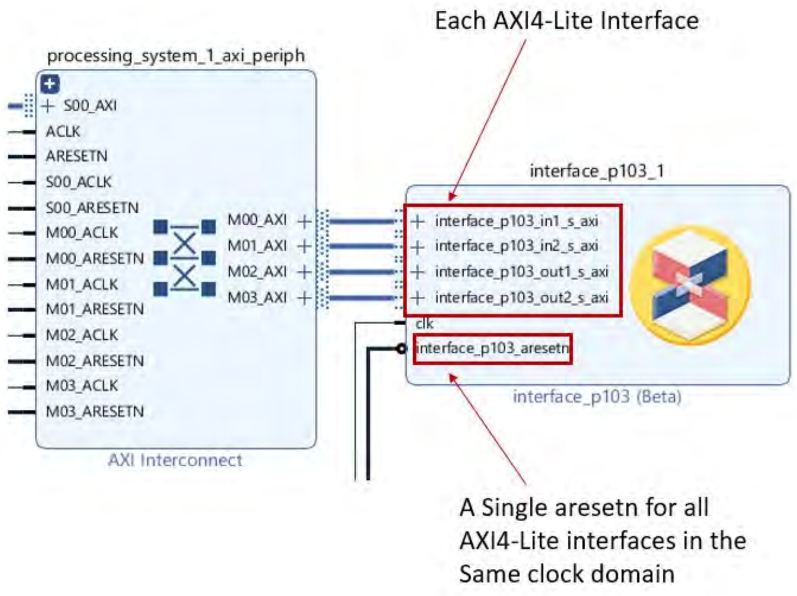


If you specify the **IP Catalog** compilation target in the System Generator token, the flow will also generate an example BD with multiple AXI4-Lite interfaces and an `aresetn` signal.

The naming convention for an interface is:

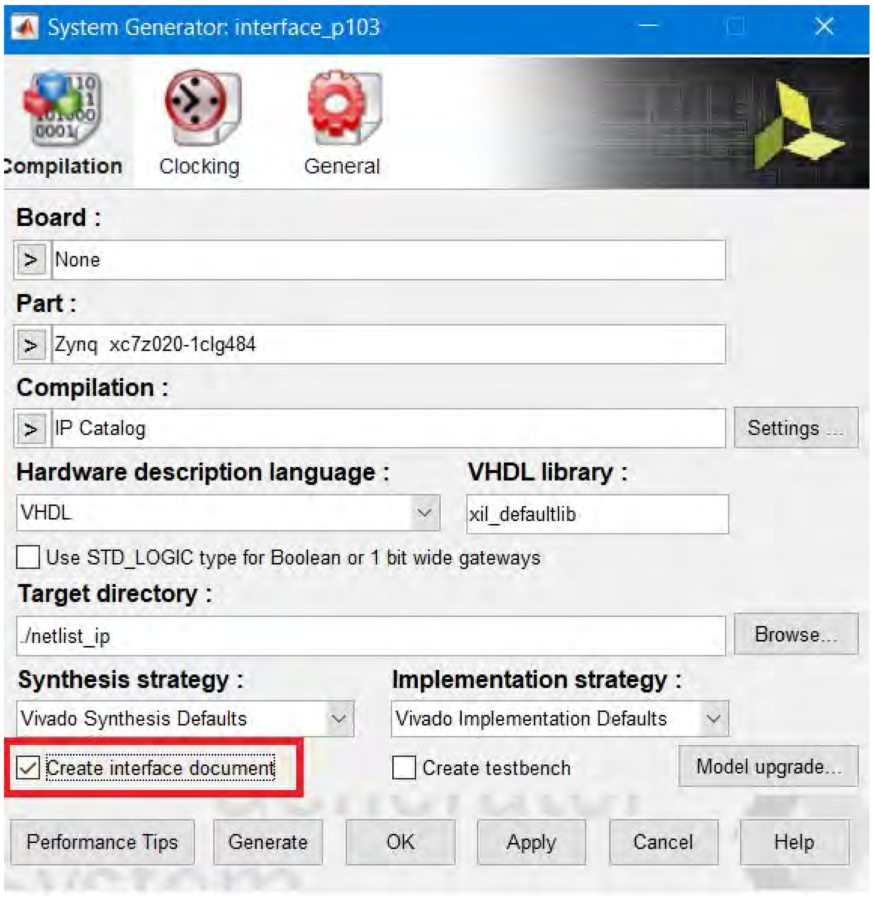
```
<clock domain name/design name>_<interface name>_s_axi
```

Figure 83: **Example BD**



To generate a document describing the IP, select the **Create interface document** option on the System Generator Token **Compilation** tab before you perform the compilation.

Figure 84: **Create Interface Document**


System Generator: interface_p103

Board : None

Part : Zynq xc7z020-1clg484

Compilation : IP Catalog

Hardware description language : VHDL

VHDL library : xil_defaultlib

☐ Use STD_LOGIC type for Boolean or 1 bit wide gateways

Target directory : ./netlist_ip

Synthesis strategy : Vivado Synthesis Defaults

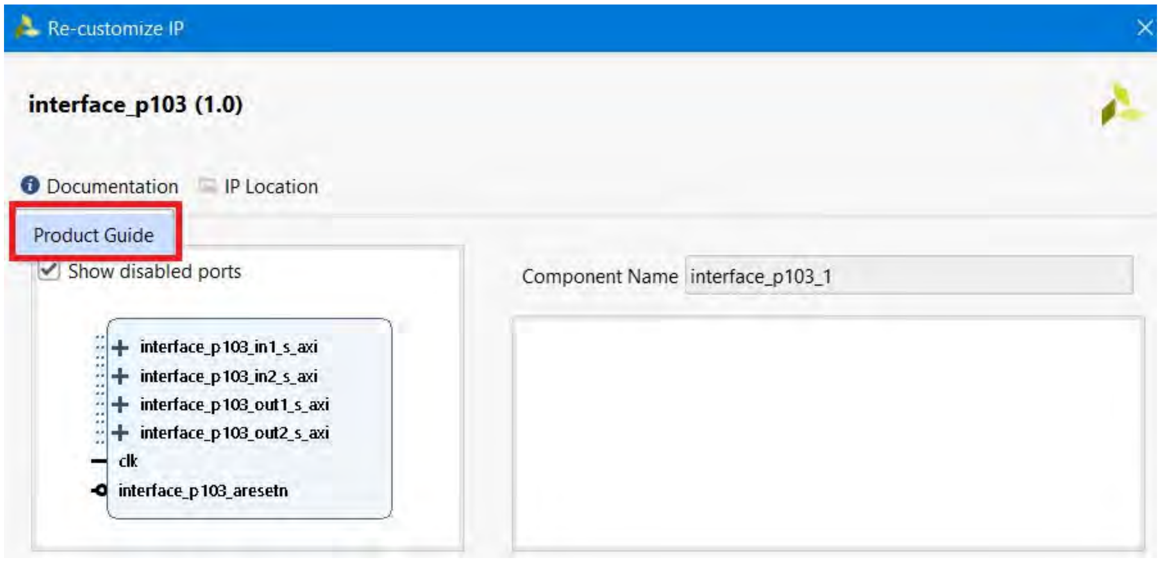
Implementation strategy : Vivado Implementation Defaults

☑ Create interface document

☐ Create testbench

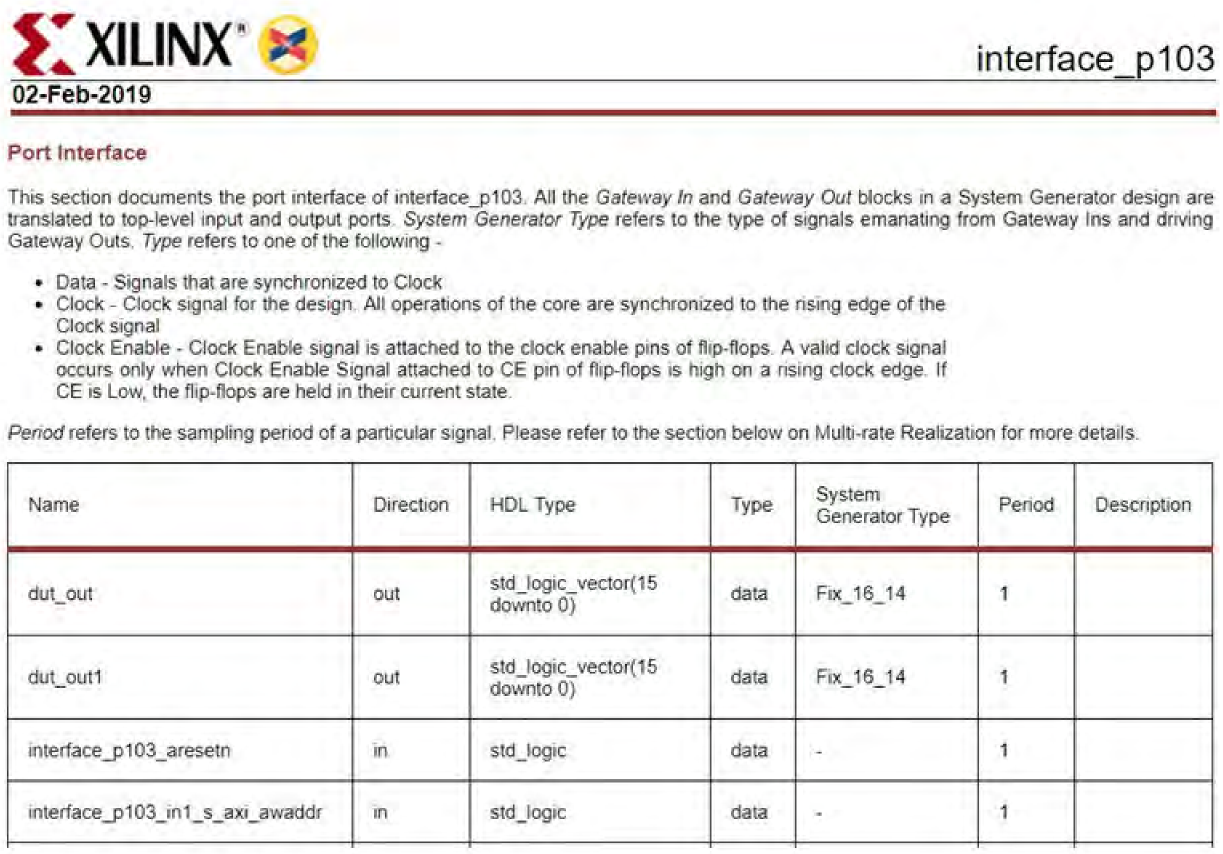
Access the document the same way you access the document for any other Vivado IP. Double-click the IP in the Vivado schematic, then select **Documentation → Product Guide**.

Figure 85: **Accessing Documents**



A document (HTML file) will open up (see example below).

Figure 86: **Sample Document**



interface_p103

02-Feb-2019

Port Interface

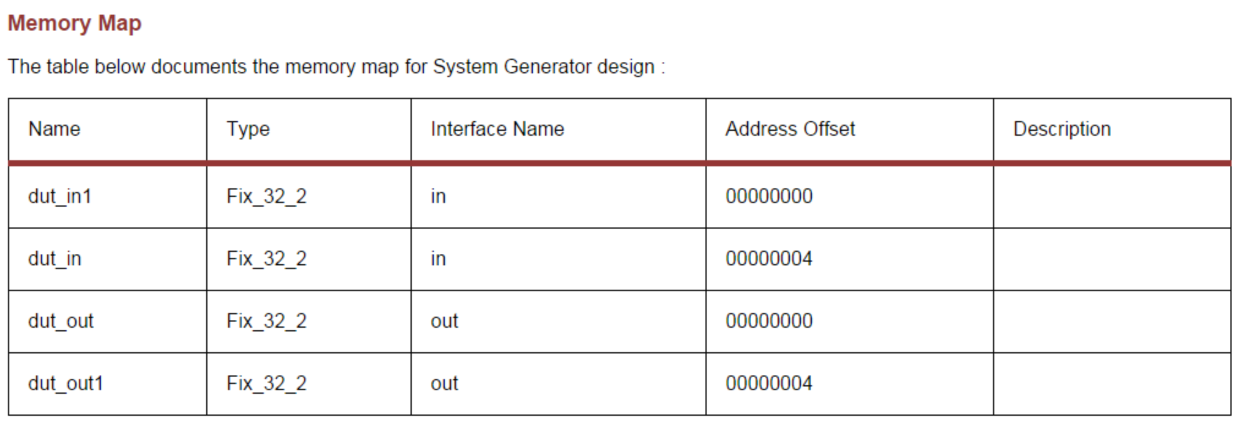
This section documents the port interface of interface_p103. All the *Gateway In* and *Gateway Out* blocks in a System Generator design are translated to top-level input and output ports. *System Generator Type* refers to the type of signals emanating from Gateway Ins and driving Gateway Outs. *Type* refers to one of the following -

- Data - Signals that are synchronized to Clock
- Clock - Clock signal for the design. All operations of the core are synchronized to the rising edge of the Clock signal
- Clock Enable - Clock Enable signal is attached to the clock enable pins of flip-flops. A valid clock signal occurs only when Clock Enable Signal attached to CE pin of flip-flops is high on a rising clock edge. If CE is Low, the flip-flops are held in their current state.

Period refers to the sampling period of a particular signal. Please refer to the section below on Multi-rate Realization for more details.

Name	Direction	HDL Type	Type	System Generator Type	Period	Description
dut_out	out	std_logic_vector(15 downto 0)	data	Fix_16_14	1	
dut_out1	out	std_logic_vector(15 downto 0)	data	Fix_16_14	1	
interface_p103_aresetn	in	std_logic	data	-	1	
interface_p103_in1_s_axi_awaddr	in	std_logic	data	-	1	

This document contains a section on the Memory Map for the IP. If you selected **Auto assign address offset** in the Gateway In or Gateway Out port for the AXI4-Lite interfaces, you can find out the address offset the different interfaces are mapped to.



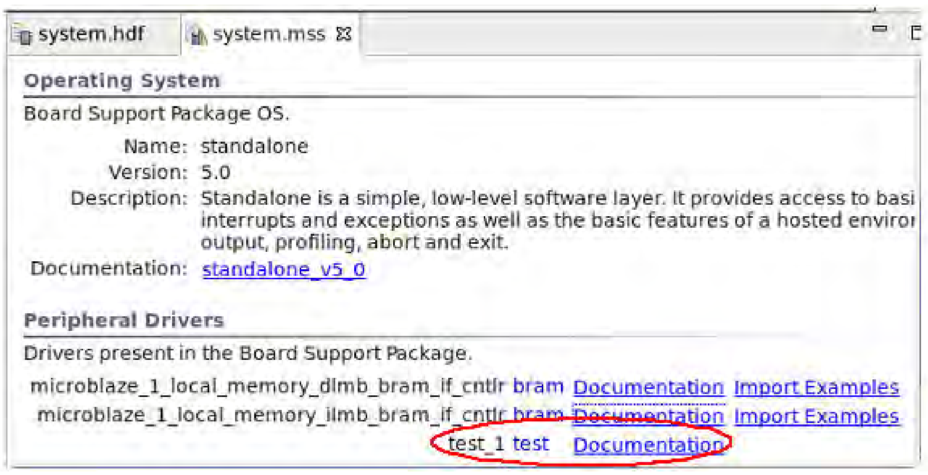
Memory Map

The table below documents the memory map for System Generator design :

Name	Type	Interface Name	Address Offset	Description
dut_in1	Fix_32_2	in	00000000	
dut_in	Fix_32_2	in	00000004	
dut_out	Fix_32_2	out	00000000	
dut_out1	Fix_32_2	out	00000004	

Software Drivers are automatically generated and packaged as well in the Vitis™ software plaform. The documentation for the software drivers can be found in the Vitis environment.

Figure 87: **Software Driver Documentation**



Operating System

Board Support Package OS.

Name: standalone
Version: 5.0
Description: Standalone is a simple, low-level software layer. It provides access to basi
interrupts and exceptions as well as the basic features of a hosted enviro
output, profiling, abort and exit.
Documentation: standalone_v5_0

Peripheral Drivers

Drivers present in the Board Support Package.

microblaze_1_local_memory_dlmb_bram_if_cntlr	bram	Documentation	Import Examples
microblaze_1_local_memory_ilmb_bram_if_cntlr	bram	Documentation	Import Examples
test_1	test	Documentation	

Address Generation

The following assumptions are made in the automatic address-generation process:

1. Each AXI4-Lite gateway is associated with a unique address offset that is aligned with a 32-bit word boundary (i.e., will be a multiple of 4).
2. Addressing begins at zero.
3. Addressing is incrementally assigned in the lexicographical order of the gateways. In the event two gateways have the same name - disambiguation will be arbitrary.
4. All AXI4-Lite gateways must be less than 32-bits wide else an error is issued.

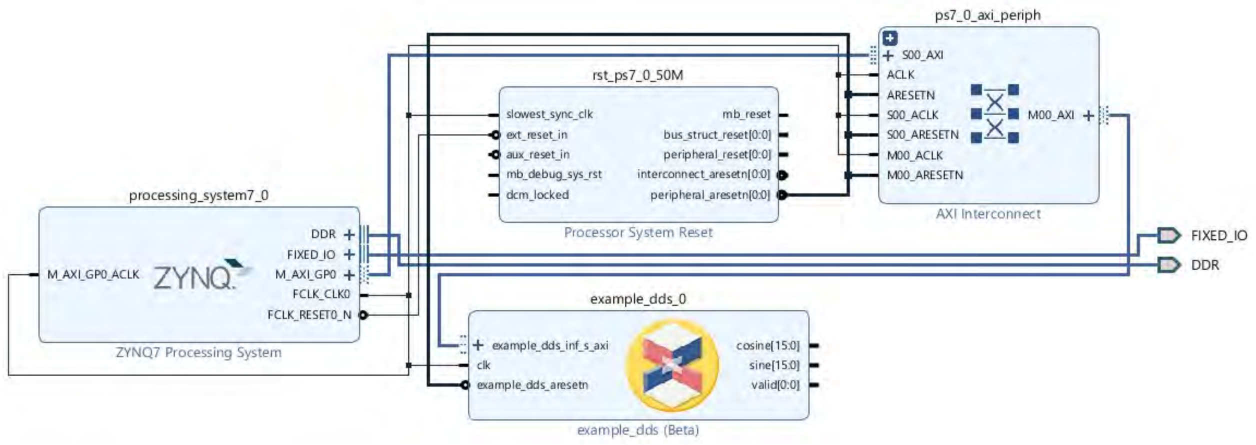
5. If an AXI4-Lite gateway is less than 32-bits wide, then from the internal register, LSBs will be assigned into the Design Under Test (DUT).
6. The following criteria is used to manage the user-specified offset addresses:
   a. All user-specified addresses are allocated to AXI4-Lite gateways before automatic allocation.
   b. If two user-specified addresses are the same, an error is issued only during generation (otherwise it will be ignored).
   c. If the remaining AXI4-Lite gateways that are set to allocate address automatically, Model Composer attempts to fill the "holes" left behind by user-specified addressing.

Features of the Vivado IDE Example Project

The Vivado® IDE example project (`example_dds.xpr`) is created to help you jump start your usage of the IP created from Model Composer. This project is configured as follows:

1. The IP generated from Model Composer is already added to the IP catalog associated with the project and available for the RTL flow as well as the IP integrator-based flow.
2. The design includes an RTL instantiation of IP called example_dds_0 underneath example_dds_stub that indicates how to instance such an IP in RTL flow.
3. The design includes a test bench called example_dds_tb that also instances the same IP in RTL flow.
4. The design includes an example IP integrator diagram with a Zynq®-7000 Subsystem as the part selected in this example is a Zynq®-7000 SoC part. For all other parts, a MicroBlaze-based Subsystem is created.

Figure 88: **IP Integrator Diagram**



5. If the part selected is the same as one of the supported boards, the project is set to the first board encountered with the same part setting.
6. A wrapper instancing the block design is created and set as Top.

TIP: *The interface documentation associated with the IP is accessible through the block GUI. To access this documentation, double-click the Model Composer IP, and click the* **Documentation** *button in the GUI.*

Software Drivers

Bare-metal software drivers are created based on the address offsets assigned to the gateways. These drivers are located in the folder called `<target_directory>/ip/drivers`. `<target_directory>/ip` must be added to the Vitis™ environment search paths to use these drivers.

For each Gateway In mapped to an AXI4-Lite interface, the following two APIs are created.

```
/**
* Write to <Gateway In id> of <design name>. Assignments are LSB-justified.
*
 * @param InstancePtr is the <Gateway In id> instance to operate on.
* @param Data is value to be written to gateway <Gateway In id>.
*
* @return None.
*
* @note    <Text from Description control of the Gateway In GUI>
*
*/
void <Gateway In id>_write(example_dds *InstancePtr, u32 Data);

/**
* Read from <Gateway In id> of <design name>. Assignments are LSB-justified.
*
* @param InstancePtr is the phase_valid instance to operate on.
*
* @return u32
*
* @note    Phase Valid Port That Must Be Asserted.
*
*/
u32 <Gateway In id>_read(example_dds *InstancePtr);
```

`<Gateway In id>`: `<design_name>_<gateway_name>` where `<design_name>` is the VHDL/Verilog top-level name of the design and `<gateway_name>` is the scrubbed name of the gateway.

Gateway Outs generate a similar driver, but are read-only.

Known Issue in AXI4-Lite Interface Generation

Test Bench generation is not supported for designs that have gateways (Gateway In or Gateway Out) configured as an AXI4-Lite Interface.

Tailor Fitting a Platform Based Accelerator Design in Model Composer

Platform based accelerators use a bottom-up design methodology to ease the development of larger systems. Two distinct design portions are created: the connectivity platform which connects board level interfaces to a processing system, and the differentiated logic accelerator(s) which represent the data path internal to the SoC and are controlled and/or fed by the connectivity platform design. DSP data paths or accelerators can take advantage of automation to tie into the connectivity platform and its interfaces to external devices.

To speed up creating a design in the Vivado IP Integrator in which the accelerator portion of the design will be developed in Model Composer, the following procedure can be used:

1. Create a Block Diagram (BD) of your design in the Vivado IP Integrator. This will act as your connectivity platform.
2. Import the connectivity platform into Model Composer.
3. Enter the accelerator portion of the design in Model Composer.
4. In Model Composer, compile the accelerator model using the IP Catalog flow, to create a Vivado project containing the original design (from the Vivado BD file) and the circuitry in the Model Composer model.

Step 1: Create a Connectivity Platform in Vivado as an IP Integrator Block Diagram (.bd)

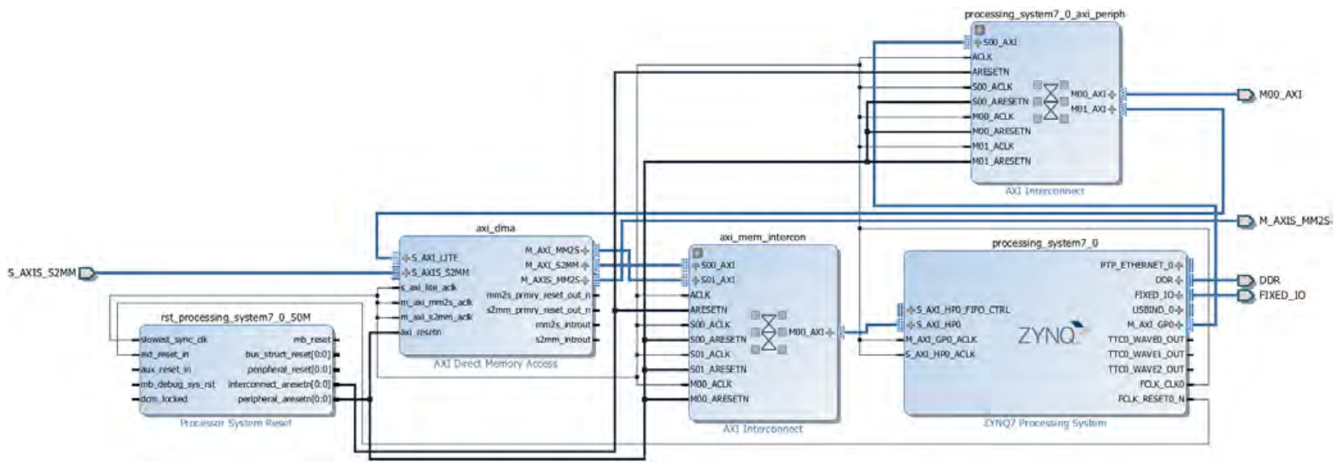
First, you must create a block diagram containing your platform design in the Vivado® IP integrator. You may use a configurable example design, a reference design, or a custom-built design as the platform based system that will contain the accelerator part of the design.

In the example below, the platform design contains a Zynq®-7000 Processing System, and AXI DMA. The connectivity platform designer intends to transfer data to and from the DDR memory using the DMA, perform DES Encryption on the data received from the DDR, and then send the encrypted data back into the DDR. The AXI4-Stream ports M_AXIS_MM2S and S_AXIS_S2MM (Data Path) are made external to the Block Diagram (BD). It shows the intent of the platform designer that these interfaces are available for Model Composer to use during the Model Composer BD import process. An AXI4-Lite interface, M00_AXI, is also made external, indicating that there will be a control interface on the accelerator IP.

These are requirements for the design in the IP integrator:

- This system has to be built for a specific board or part. This ensures that certain ports and interfaces have known location attributes assigned to them.
- The AXI Interfaces that you want to bring into the accelerator portion of the design have to be made external.

Figure 89: **AXI Interfaces**



Currently we support the following interfaces from the platform framework point of view:

Table 4: **Supported AXI Interfaces**

Interface	Master	Slave
AXI4	Yes	No
AXI4-Lite	Yes	No
AXI4-Stream	Yes	Yes

Step 2: Parse the BD File and Import Un-Located Ports and Interfaces into Model Composer

You can now use the `xilinx.utilities.importBD` utility in Model Composer to import the BD (Block Diagram) that you created in the Vivado® IP integrator.

This utility takes in the platform framework Vivado project and the name of the new model to be created in Model Composer. It parses the platform design for potential Model Composer ports and external interfaces (that is, interfaces whose ports do not have location attributes, based on the board connectivity and automation) and creates a sample stub in Model Composer representing the accelerator portion of the design.

COMMAND USAGE:

`xilinx.utilities.importBD` takes in the platform Vivado project and the name of the new model to be created. It parses the platform for potential Model Composer ports and interfaces and creates a sample stub for the user to make development easy. If the new model name is not specified an untitled model will be opened.

Inputs are: The Vivado project and the model_name (optional)

USAGE:

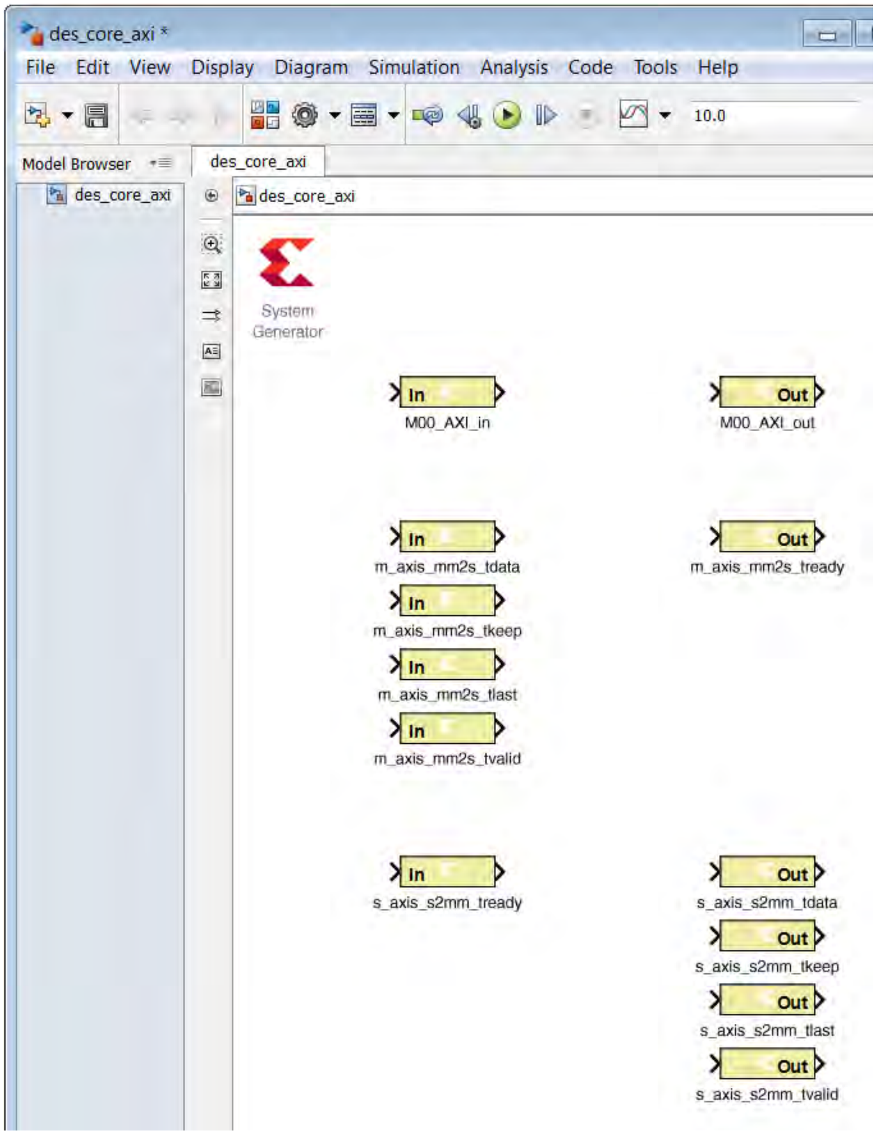
```
xilinx.utilities.importBD('<full_or_relative_path_to_vivado_project_director
y>/

<project_name>.xpr', 'mynewmodel')
```

EXAMPLES

```
xilinx.utilities.importBD('C:\test_importBD\platform.xpr', 'mynewmodel')
xilinx.utilities.importBD('C:\test_impportBD\platform.xpr')
```

In Model Composer, the resulting model will look like the example below.

Figure 90: **Model Composer Model**



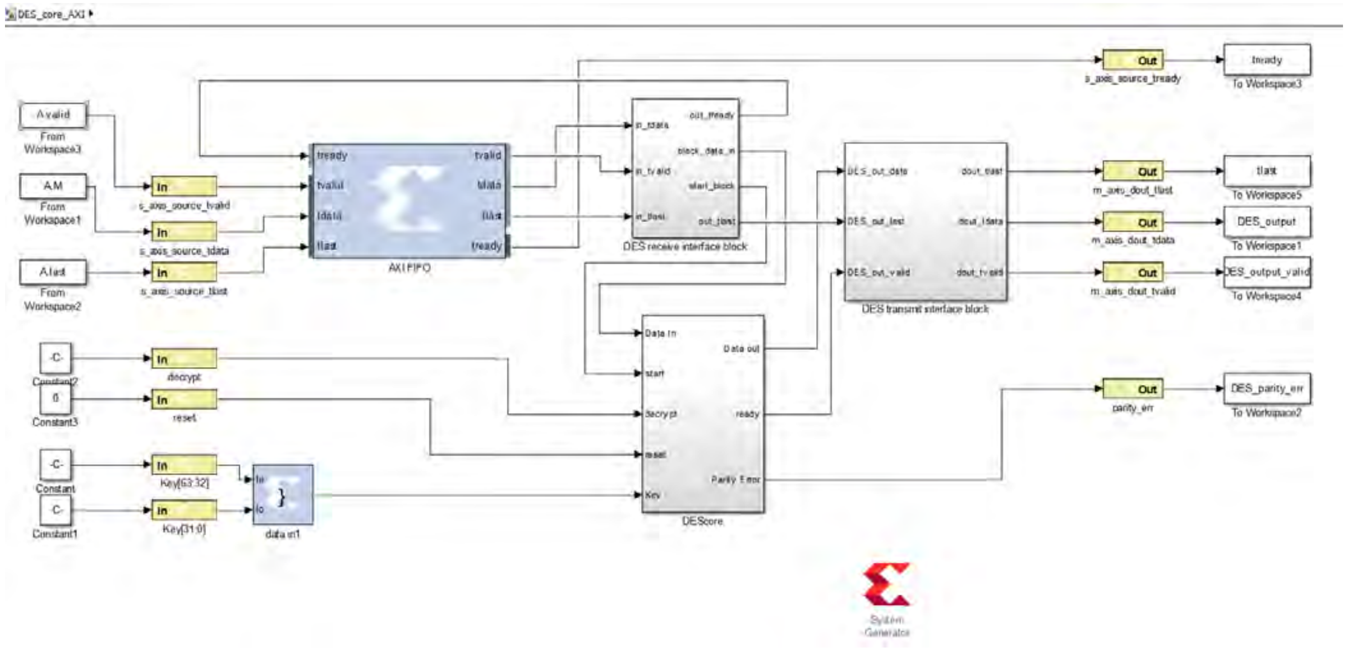
The model in Model Composer will have these features:

- For each AXI4-Lite interface, a Gateway In and a Gateway Out block will appear. You can then replicate and add as many AXI4-Lite gateways as your design requires.
- For an AXI4-Stream interface, the associated TDATA, TVALID, TREADY, and other AXI4-Stream ports will appear.
- The model's System Generator token is set to a Compilation target of **IP Catalog** and the **Part** or **Board** will be set to the same Xilinx device or board as that of the Vivado project.

Step 3: In Model Composer, Connect Logic to the BD Socket

At this point you can create the accelerator in Model Composer. In the example below we have connected to some other logic, and renamed the gateways.

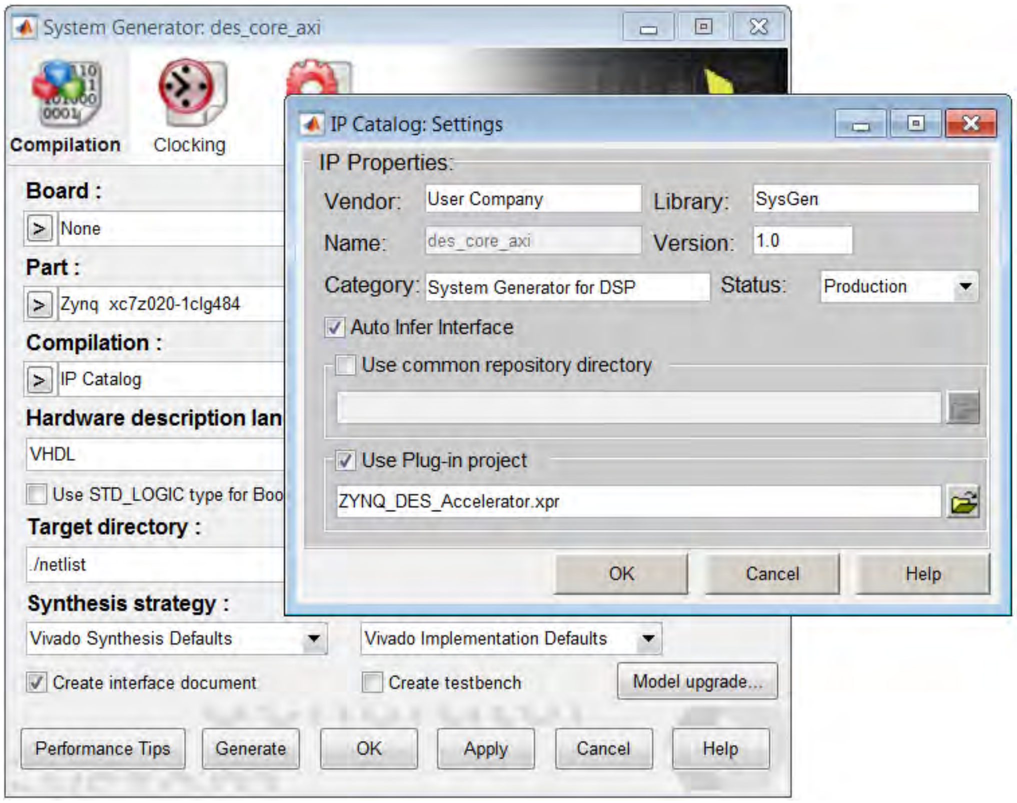
Figure 91: **Connect to BD Socket**



Step 4: Compile the Accelerator Model (IP Catalog Flow) to Create a Complete Design

You can now use the IP catalog compilation flow to create a complete design. When you double-click the System Generator token in IP catalog flow and click the **Settings** button, the **Use Plug-in project** directory must point to the Vivado® IP integrator project from which the design was imported (see below). When you click the **Generate** button, a new Vivado project based on the original Vivado platform framework/system plus the accelerator IP created in Model Composer, along with a software driver, will be created. This project will be located in an ip_catalog directory under the System Generator token's **Target directory**, and can also be placed into a common IP repository.

Figure 92: **Target Directory**

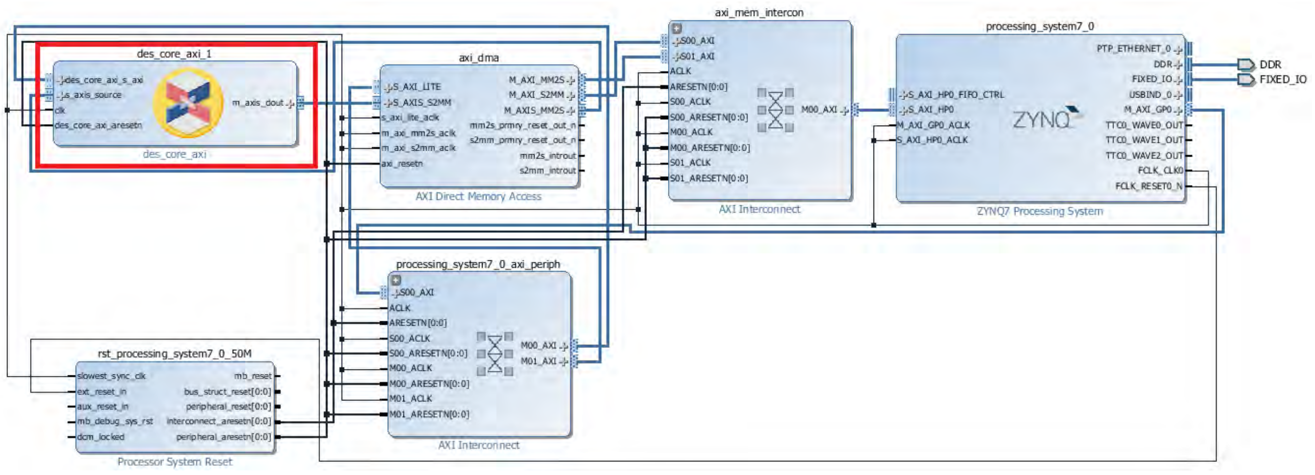


Board : None
Part : Zynq xc7z020-1clg484
Compilation : IP Catalog
Hardware description lan: VHDL
Use STD_LOGIC type for Boo
Target directory : ./netlist
Synthesis strategy : Vivado Synthesis Defaults
Vivado Implementation Defaults
☑ Create interface document
☐ Create testbench

IP Properties:
Vendor: User Company
Library: SysGen
Name: des_core_axi
Version: 1.0
Category: System Generator for DSP
Status: Production
☑ Auto Infer Interface
☐ Use common repository directory
☑ Use Plug-in project
ZYNQ_DES_Accelerator.xpr

You can open this new project in Vivado to complete the implementation of your design. The block highlighted in the following figure indicates a block developed using Model Composer HDL Blockset.

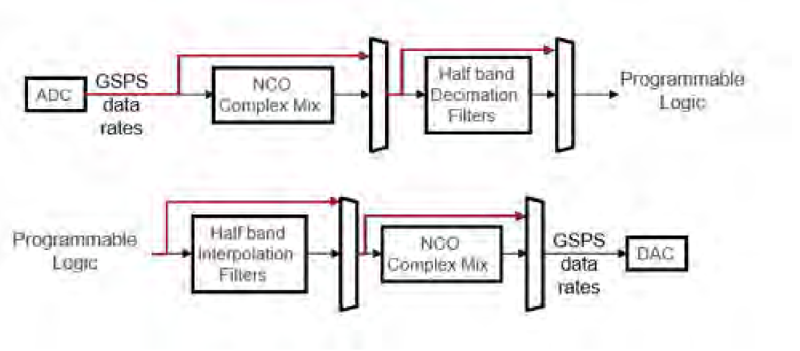
Figure 93: **New Project**



Using Super Sample Rate (SSR) Blocks in Model Composer

While the Super Sample Rate (SSR) feature introduced in this section can be widely applicable to all Xilinx® devices, this section explains the motivation for it for Xilinx RFSoC devices. The integration of direct RF-sampling data converters with Xilinx's technology offers the most flexible, smallest footprint, and lowest power solution for a wide range of high performance RF applications such as Wireless communications, cable access, test & measurement, and radar. RFSoC devices provide hardened Digital Up Converters (DUC) and Digital Down Converters (DDC). NCO, Complex Mixers, and Filters are hard Macros, and filter characteristics are optimized for general Commercial applications.

Figure 94: **RFSoC Device**



Depending on what is needed, RFSoC devices can be used in two ways.

- Use the available hardened NCO & Complex Mix and Half Band Decimation/interpolation filters.

- If the sequence of the hardened blocks does not meet the design requirement, you can bypass them as shown in the figure above.

In the latter case, to meet the design requirements, you may need to implement the NCO, Complex Mixerm and DDC blocks in the fabric using the HDL Blockset in Model Composer. To do this, bypass the hardened blocks, and let Model Composer IPs run at Programmable Logic (PL) clock frequency. When the sample rate from the ADC is in GSPS, and PL handles only the MSPS range of data, you must accept and compute multiple parallel samples every clock cycle for each data channel. The number of parallel samples is determined by calculating the ratio between the sample frequency and the Programmable Logic clock frequency, which is defined as an SSR parameter.

What is SSR?

SSR is a parameter that determines how many parallel samples to accept for every clock cycle.

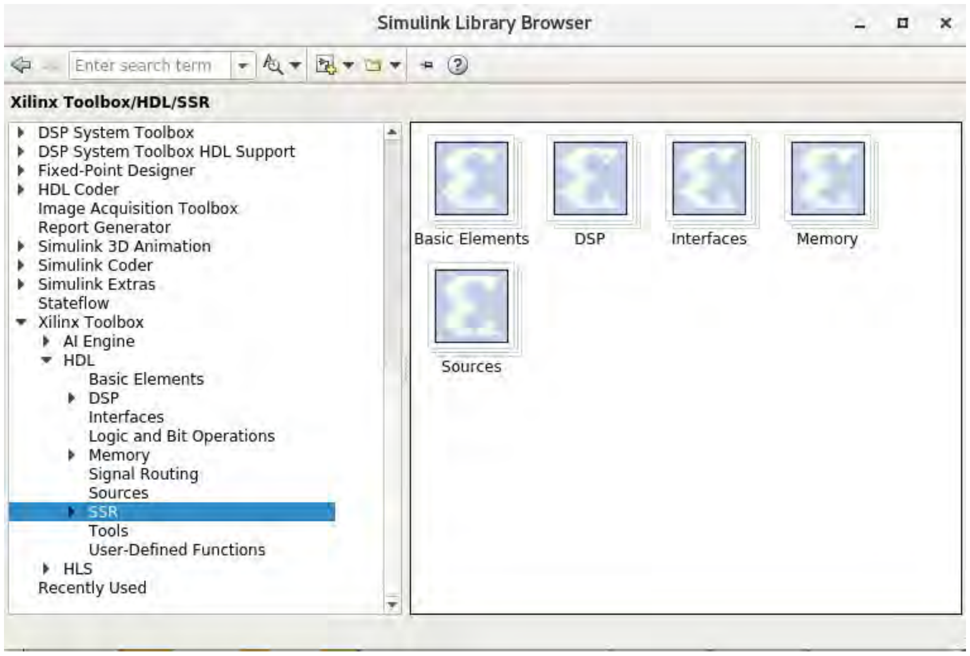
How SSR helps users?

- SSR is beneficial for users who cannot use the hardened RFSoC DUC and DDCs.
- SSR provides programmatic subsystems for NCO and Complex Mixer among many others. The user input parameters in the block mask and Model Composer programmatically construct the underlying subsystem with multiple DDS blocks.
- SSR avoids manual and structural modifications to your design, which accelerates the design-cycle.

SSR Library

Model Composer provides a separate set of library blocks for handling SSR. Currently, Model Composer supports 25 vector blocks, which can be accessed from the MATLAB® Library Browser.

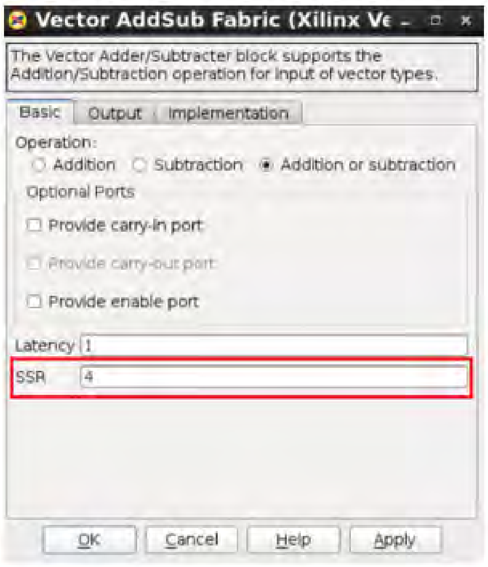
Figure 95: **SSR Block set in HDL Library**



The SSR parameter can be defined for all the blocks present in the SSR block set. When you add a block from the library, the default SSR value is 4, and the maximum SSR valiue is 256.

The SSR block set is defined in Xilinx SSR Blockset.

Figure 96: **Default SSR Value**


Vector AddSub Fabric (Xilinx Ve

The Vector Adder/Subtracter block supports the Addition/Subtraction operation for input of vector types.

Operation:
Addition ( ) Subtraction ( ) [✓ Addition or subtraction]

Optional Ports

☐ Provide carry-in port

☐ Provide carry-out port

☐ Provide enable port

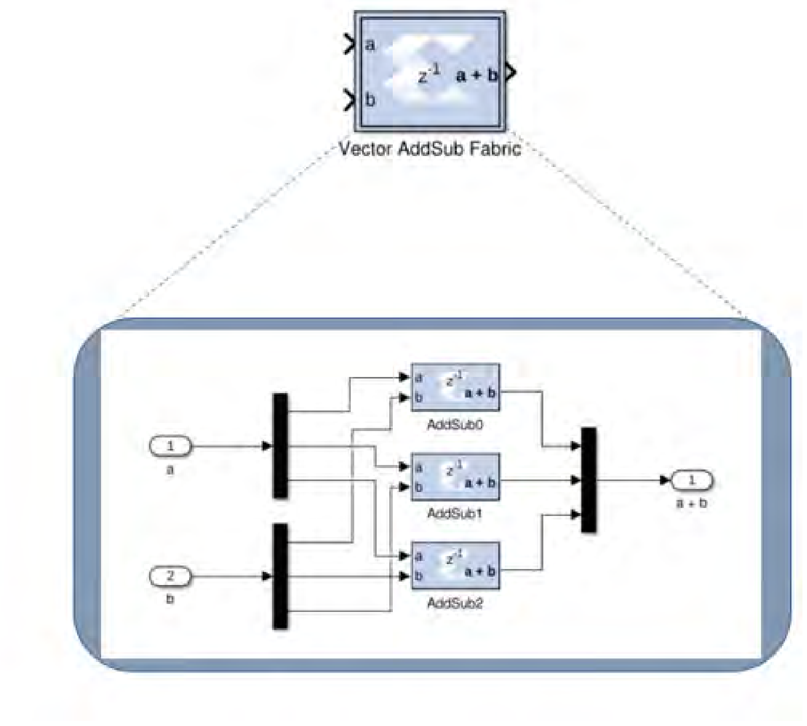
Latency: 1

SSR: 4

No matter what the SSR rate is, you only need to provide a limited number of signal connections as with a normal IP block. Model Composer automatically takes care of all the parallel path connection internal to the SSR block, according to the SSR parameter value provided.

For example, for a Vector AddSub block, when SSR parameter is modified to 3, the internal connections are done automatically as shown below. This creates 3 parallel paths for computation and results in single output.

Figure 97: **Vector AddSub Fabric Example**



Performing Analysis in Model Composer

Model Composer is a bit and cycle accurate modeling tool. You can verify the functionality of your designs by simulating in Simulink®. However, to ensure that your Model Composer design will work correctly when it is implemented in your target Xilinx® device, these analysis tools have been integrated into Model Composer:

- **Timing Analysis:** To ensure that the HDL files generated by Model Composer operate correctly in hardware, you must close timing. To help accelerate this process, timing analysis has been integrated into Model Composer.

- **Resource Analysis:** To ensure that the HDL files generated by Model Composer will fit into your target device, you may need to analyze the resources being used. To help accelerate this process, resource analysis has been integrated into Model Composer.

Timing Analysis in Model Composer	Presents an overview of timing analysis in Model Composer.
Performing Timing Analysis	Describes how to perform timing analysis on your model.
Cross Probing from the Timing Analysis Results to the Model	Describes how you can cross probe from a row in the Timing Analyzer table to the Simulink model, highlighting the corresponding HDL blocks in the path.
Accessing Existing Timing Analysis Results	Describes how to re-launch the Timing Analyzer table on pre-existing Timing Analysis results.
Recommendations For Troubleshooting Timing Violations	Describes methods to help you discover the source of timing violations in your design.
Resource Analysis in Model Composer	Presents an overview of resource analysis in Model Composer.
Performing Resource Analysis	Describes how to perform resource analysis on your model.
Cross Probing from the Resource Analysis Results to the Model	Describes how you can cross probe from a row in the Resource Analyzer table to the Simulink model, highlighting the corresponding block or subsystem in the design.
Accessing Existing Resource Analysis Results	Describes how to re-launch the Resource Analyzer table on pre-existing Resource Analysis results.
Recommendations for Optimizing Resource Analysis	Describes methods to help you use the Resource Analyzer to optimize resource utilization in the design.

Timing Analysis in Model Composer

To ensure that the HDL files generated by Model Composer work correctly in hardware, you must close timing. To help accelerate this process, timing analysis has been integrated into Model Composer.

Timing analysis allows you to perform static timing analysis on the HDL files generated from Model Composer, either Post-Synthesis or Post-Implementation. It also provides a mechanism to correlate the results of running the Vivado® Timing Engine on either the Post-Synthesized netlist or the Post Implementation netlist with the Model Composer model in Simulink®. Thus, you do not have to leave the Simulink® modeling environment to close timing on the DSP sub-module of the design.

Invoking timing analysis on a compilation target (for example, HDL Netlist) results in a tabulated display of paths with columns showing information such as timing slack, path delay, etc. This is the Timing Analyzer table. You can sort the contents of the table using any of the column metrics such as slack, etc. Also, cross probing is enabled between the table entries and the Simulink model to accelerate finding and fixing timing failures in the model. Cross probing between the Timing Analyzer table and the Simulink model is accomplished by selecting/clicking a row in the table. The corresponding path in the model will be highlighted. The path is highlighted in red if the path corresponds to a timing violation; otherwise it is highlighted in green.

Performing Timing Analysis

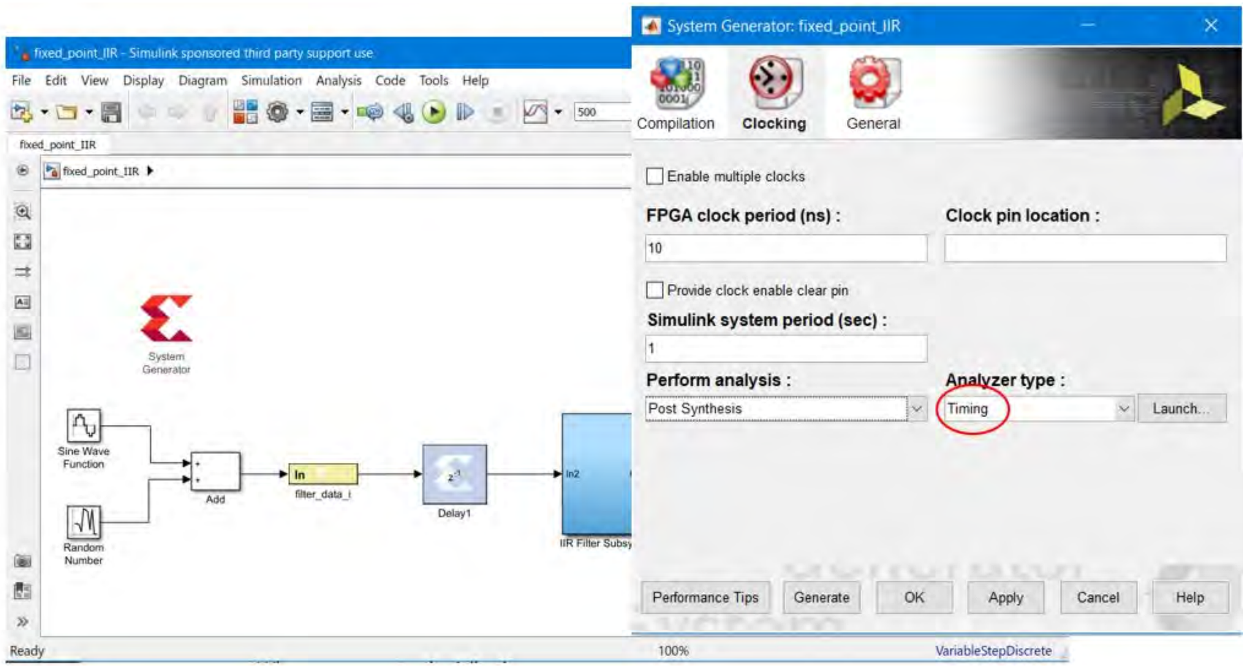
Timing analysis can be invoked whenever you generate any of the following compilation targets:

- IP catalog
- Hardware Co-Simulation
- Synthesized Checkpoint
- HDL Netlist

To perform timing analysis in Model Composer:

1. Double-click the System Generator token in the Simulink model.
2. Enter the following in the System Generator token dialog box:
    - In the **Compilation** tab, specify a **Target Directory**.
    - In the **Clocking** tab, set the **Perform Analysis** field to **Post Synthesis** or **Post Implementation** based on the runtime vs. accuracy tradeoff.
    - In the **Clocking** tab, set the **Analyzer Type** field to **Timing**.

Figure 98: **Performing Timing Analysis**



Enable multiple clocks: unchecked
FPGA clock period (ns) : 10
Clock pin location :
Provide clock enable clear pin: unchecked
Simulink system period (sec) : 1
Perform analysis : Post Synthesis
Analyzer type : Timing

3. In the System Generator token dialog box, click **Generate**.

   When you generate, the following occurs:

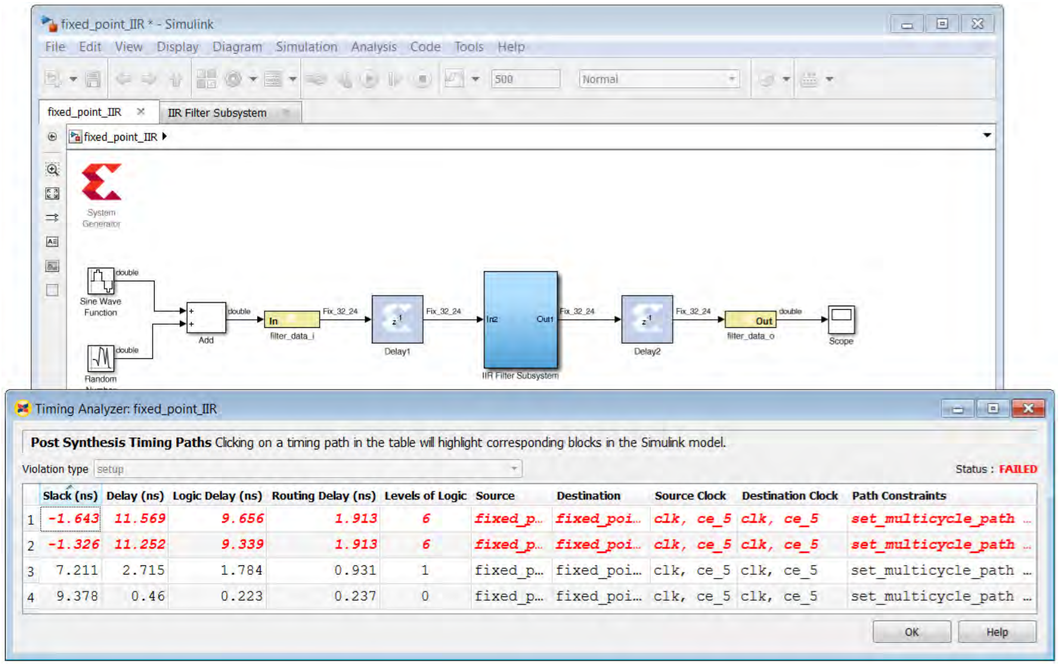
   a. Model Composer generates the required files for the selected compilation target. For timing analysis Model Composer invokes Vivado in the background for the design project, and passes design timing constraints to Vivado.

   b. Depending on your selection for **Perform Analysis** (**Post Synthesis** or **Post Implementation**), the design runs in Vivado through synthesis or through implementation.

   c. After the Vivado tools run is completed, timing paths information is collected and saved in a specific file format from the Vivado timing database. At the end of the timing paths data collection the Vivado project is closed and control is passed to the MATLAB®/Model Composer process.

   d. Model Composer processes the timing information and displays a Timing Analyzer table with timing paths information (see below).

Figure 99: **Timing Analyzer Table**



Post Synthesis Timing Paths Clicking on a timing path in the table will highlight corresponding blocks in the Simulink model.

Violation type: setup

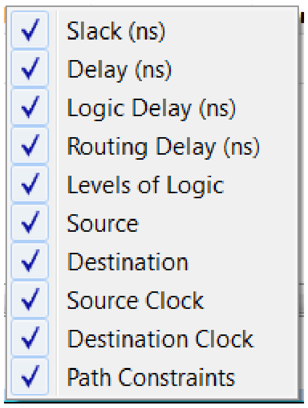
Status : FAILED

	Slack (ns)	Delay (ns)	Logic Delay (ns)	Routing Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Clock	Path Constraints
1	-1.643	11.569	9.656	1.913	6	fixed_p...	fixed_poi...	clk, ce_5	clk, ce_5	set_multicycle_path ...
2	-1.326	11.252	9.339	1.913	6	fixed_p...	fixed_poi...	clk, ce_5	clk, ce_5	set_multicycle_path ...
3	7.211	2.715	1.784	0.931	1	fixed_p...	fixed_poi...	clk, ce_5	clk, ce_5	set_multicycle_path ...
4	9.378	0.46	0.223	0.237	0	fixed_p...	fixed_poi...	clk, ce_5	clk, ce_5	set_multicycle_path ...

In the timing analyzer table:

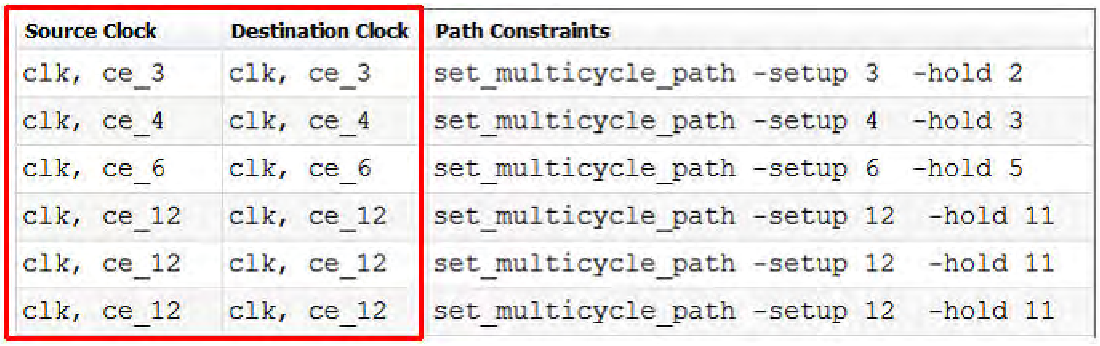
- Only unique paths from the Simulink model are reported.
- The 50 paths with the lowest Slack values are displayed with the worst Slack at the top, and increasing Slack below.
- Paths with timing violations have a negative Slack and display in red.
- The display order can be sorted for any column's values by clicking the column head.
- If you want to hide a column in the table, right-click any column head in the table and deselect the column to hide in the list that appears.

Figure 100: **Hide/Show Dialog**



- For a design with multiple clock cycle constraints, the Timing Analyzer can identify multicycle path constraints, and show them in the **Path Constraints** column. In that case, the **Source Clock**, and **Destination Clock** columns display clock enable signals to reflect different sampling rates.

Figure 101: **Clock Enable Signals**



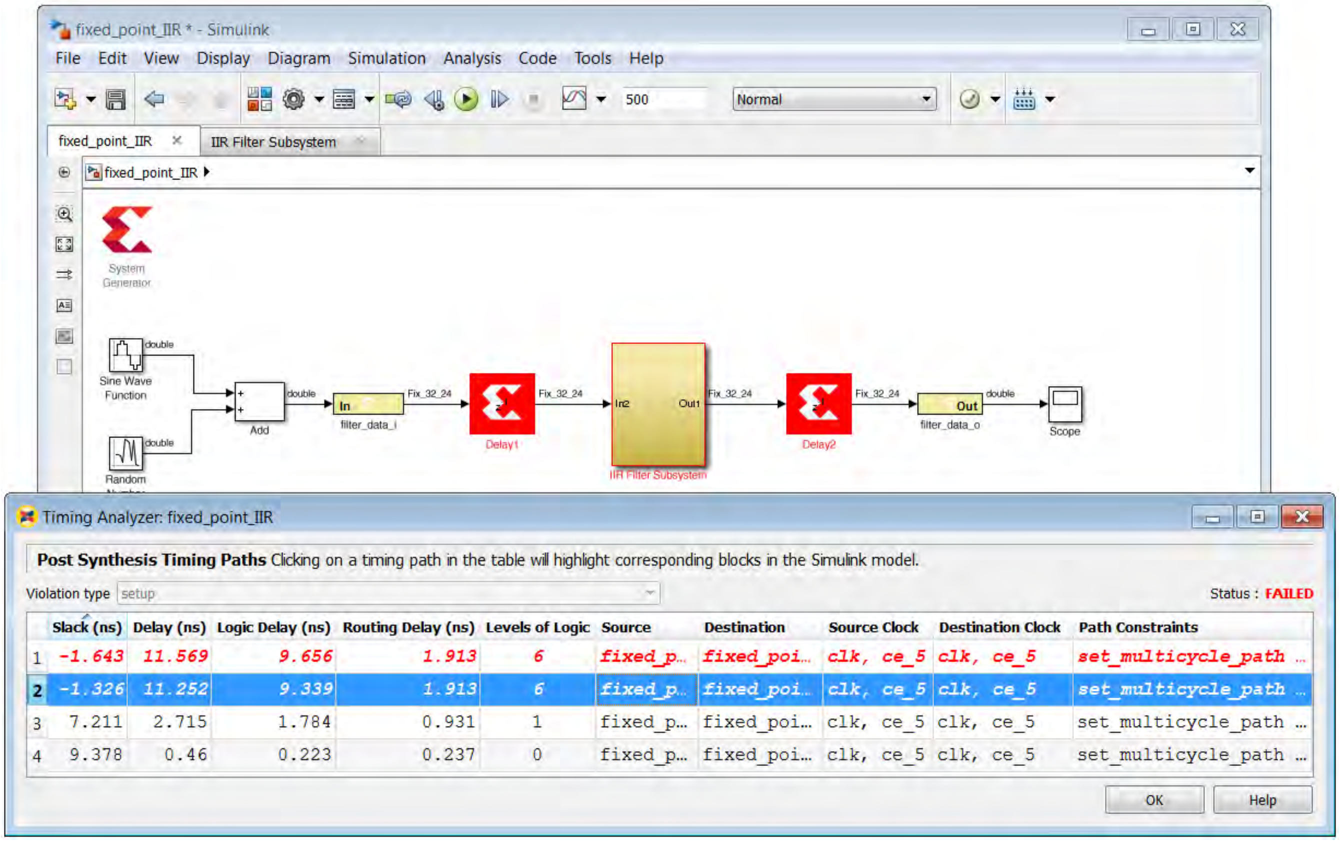
Source Clock	Destination Clock	Path Constraints
clk, ce_3	clk, ce_3	set_multicycle_path -setup 3 -hold 2
clk, ce_4	clk, ce_4	set_multicycle_path -setup 4 -hold 3
clk, ce_6	clk, ce_6	set_multicycle_path -setup 6 -hold 5
clk, ce_12	clk, ce_12	set_multicycle_path -setup 12 -hold 11
clk, ce_12	clk, ce_12	set_multicycle_path -setup 12 -hold 11
clk, ce_12	clk, ce_12	set_multicycle_path -setup 12 -hold 11

- You can cross probe from the table to the Simulink model by selecting a path in the table, which will highlight the corresponding HDL blocks in the Simulink model. See Cross Probing from the Timing Analysis Results to the Model.

Cross Probing from the Timing Analysis Results to the Model

You can cross probe from the Timing Analyzer table to the Simulink model by clicking any path (row) in the Timing Analyzer table, which highlights the corresponding HDL blocks in the model. This allows you to troubleshoot timing violations by analyzing the path on which they occur.

Figure 102: **Timing Analyzer Table**



Post Synthesis Timing Paths Clicking on a timing path in the table will highlight corresponding blocks in the Simulink model.

Violation type: setup

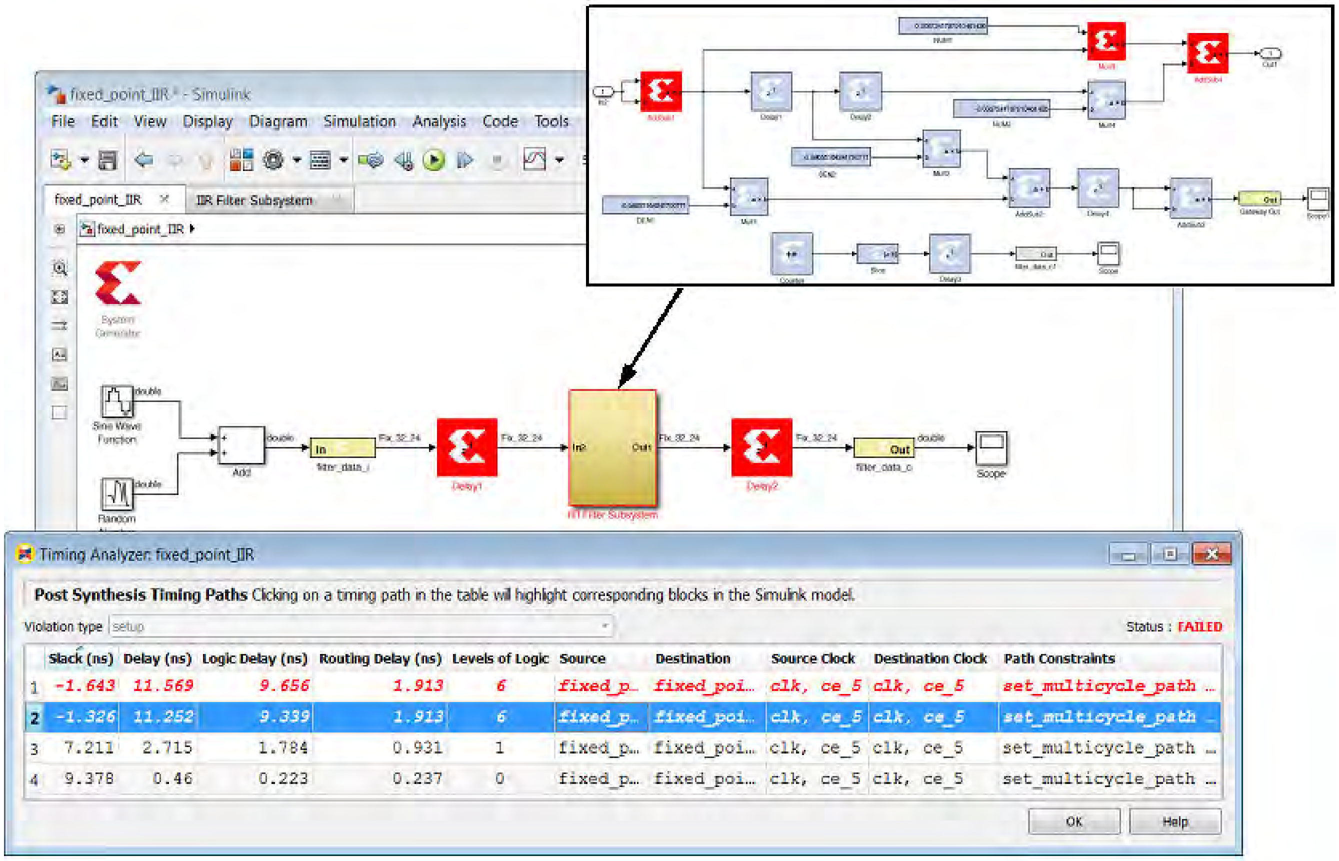
Status : FAILED

	Slack (ns)	Delay (ns)	Logic Delay (ns)	Routing Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Clock	Path Constraints
1	-1.643	11.569	9.656	1.913	6	fixed_p...	fixed_poi...	clk, ce_5	clk, ce_5	set_multicycle_path ...
2	-1.326	11.252	9.339	1.913	6	fixed_p...	fixed_poi...	clk, ce_5	clk, ce_5	set_multicycle_path ...
3	7.211	2.715	1.784	0.931	1	fixed_p...	fixed_poi...	clk, ce_5	clk, ce_5	set_multicycle_path ...
4	9.378	0.46	0.223	0.237	0	fixed_p...	fixed_poi...	clk, ce_5	clk, ce_5	set_multicycle_path ...

When you cross probe, the following will display in the model:

- Blocks in a path with a timing violation are highlighted in red in the model, whereas blocks that belong to a path with no timing violation (that is, a path with a positive Slack value) are highlighted in green in the model.
- If blocks in a highlighted path are inside a subsystem, then the subsystem is highlighted in red so you may expand the subsystem to inspect the blocks underneath.

Figure 103: **Cross Probing**



Post Synthesis Timing Paths Clicking on a timing path in the table will highlight corresponding blocks in the Simulink model.

Violation type setup

Status : FAILED

	Slack (ns)	Delay (ns)	Logic Delay (ns)	Routing Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Clock	Path Constraints
1	-1.643	11.569	9.656	1.913	6	fixed_p...	fixed_poi...	clk, ce_5	clk, ce_5	set_multicycle_path ...
2	-1.326	11.252	9.339	1.913	6	fixed_p...	fixed_poi...	clk, ce_5	clk, ce_5	set_multicycle_path ...
3	7.211	2.715	1.784	0.931	1	fixed_p...	fixed_poi...	clk, ce_5	clk, ce_5	set_multicycle_path ...
4	9.378	0.46	0.223	0.237	0	fixed_p...	fixed_poi...	clk, ce_5	clk, ce_5	set_multicycle_path ...

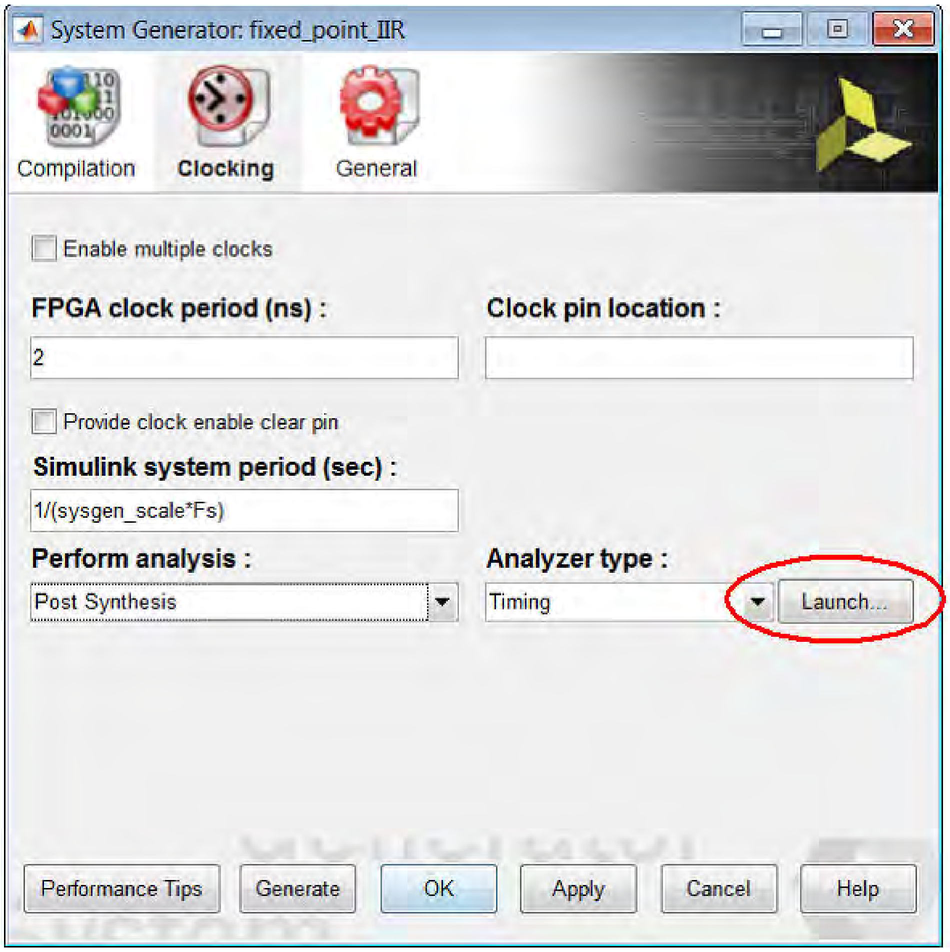
- When you select a path (row in the table) to cross probe, this normally highlights the destination block at the end of the path. That brings the subsystem containing the destination block to the front in the model. As a result, you may not be able to see the highlighted source block if the source block is in a different subsystem. If you want to see the source block, click the path in the **Source** column in the table. This will bring the subsystem containing the source block to the front of the model. Selecting the path in any other column will bring the subsystem containing the destination block to the front.

Accessing Existing Timing Analysis Results

A **Launch** button is provided under the **Clocking** tab of the System Generator token dialog box to relaunch the Timing Analyzer table using the existing timing analysis results for the model. Make sure the **Target directory** specified on the **Compilation** tab of the dialog box is readable by the Timing Analyzer, and the **Analyzer Type** field is set to **Timing**. This will only work if you already ran timing analysis on the Simulink model and haven't changed the Simulink model since the last run.

When you click the **Launch** button, the Timing Analyzer table will display the timing analysis results stored in the specified **Target directory**, regardless of the option selected for **Perform analysis** (**Post Synthesis** or **Post Implementation**).

Figure 104: **Launch Button**


System Generator: fixed_point_IIR

Enable multiple clocks: unchecked

FPGA clock period (ns) : 2

Clock pin location :

Provide clock enable clear pin: unchecked

Simulink system period (sec) : 1/(sysgen_scale*Fs)

Perform analysis : Post Synthesis

Analyzer type : Timing

You can also launch the Timing Analyzer table to display existing timing analysis results by entering this command at the MATLAB® command prompt:

```
xlAnalyzeTiming(<mdl_hdle>, <netlist_dir>)
```

where `<mdl_hdle>` is the Simulink® model handle (the handle of the top level design), and `<netlist_dir>` is the **Target directory** specified in the System Generator token dialog box.

Recommendations For Troubleshooting Timing Violations

The following are recommended for troubleshooting timing violations:

- For quicker timing analysis iterations, post-synthesis analysis is preferred over post-implementation analysis.

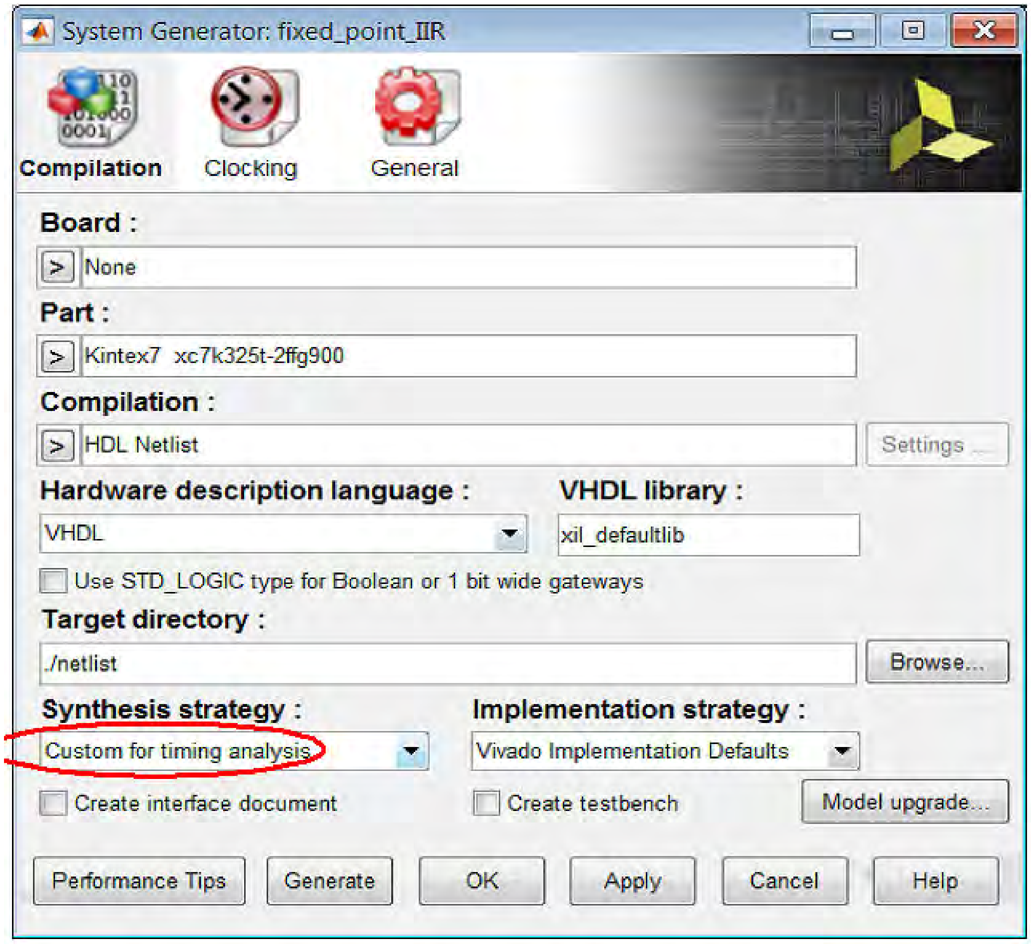
- After logic optimization during the Vivado Synthesis process the tool doesn't keep information about merged logic in the Vivado database. Merged and shared logic may make it difficult to accurately cross probe from Vivado timing paths to the Simulink model. Hence, it is recommended that you create a custom Vivado Synthesis strategy to control merged and shared logic.

For information about how to create a custom Synthesis strategy in Vivado, see this link in the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893).

To control merged and shared logic in the Vivado IDE, make the following changes to the default Vivado Synthesis strategy.

1. Set these Synthesis options in Vivado IDE:
   - Select the Synthesis option `-keep_equivalent_registers`.
   - Set the Synthesis option `-resource_sharing` to the value `off`.
2. Save the new Synthesis strategy and exit Vivado IDE.
3. In Model Composer, select the new custom **Synthesis strategy** in the System Generator token dialog box before generating the design.

Figure 105: **Custom for Timing Analysis**


System Generator: fixed_point_IIR

Board : None

Part : Kintex7 xc7k325t-2ffg900

Compilation : HDL Netlist

Hardware description language : VHDL

VHDL library : xil_defaultlib

Use STD_LOGIC type for Boolean or 1 bit wide gateways

Target directory : ./netlist

Synthesis strategy : Custom for timing analysis

Implementation strategy : Vivado Implementation Defaults

Create interface document

Create testbench

Resource Analysis in Model Composer

To ensure that the HDL files generated by Model Composer will fit into your target device, you may need to analyze the resources being used. To help accelerate this process, resource analysis has been integrated into Model Composer.

Resource analysis allows you to determine the number of look-up tables (LUTs), registers, DSP48s (DSPs), and block RAMs (BRAMs) used by your model. The analysis is performed either Post-Synthesis or Post-Implementation and provides a mechanism to correlate the resources used in the Vivado® tools with the Model Composer model in Simulink®. Thus, you do not have to leave the Simulink modeling environment to investigate and determine areas where excessive resources are being used in your design.

Invoking resource analysis on a compilation target (for example, IP catalog) results in a tabulated display of blocks, and hierarchies showing LUT, Register, DSP, and block RAM resource usage. This is the Resource Analysis table. You can sort the contents of the table using any of the column metrics such as DSPs, etc. Also, cross probing is enabled between the table entries and the Simulink model to accelerate finding and fixing excessive resource usage in the model. Cross probing between the Resource Analysis table, and the Simulink model is accomplished by selecting (clicking) a row in the table. The corresponding block, or hierarchy in the model is highlighted in yellow.

Performing Resource Analysis

Resource analysis can be performed whenever you generate any of the following compilation targets:

- IP Catalog
- Hardware Co-Simulation
- Synthesized Checkpoint
- HDL Netlist

To perform resource analysis in Model Composer:

1. Double-click the System Generator token in the Simulink model.
2. Select the following in the System Generator token dialog box:
   a. In the **Compilation** tab:
      - Specify the **Part** in which your design will be implemented.

        ***Note*: If you select a **Board** instead of a **Part**, the **Part** field will be filled in with the name of the part on the selected **Board**.

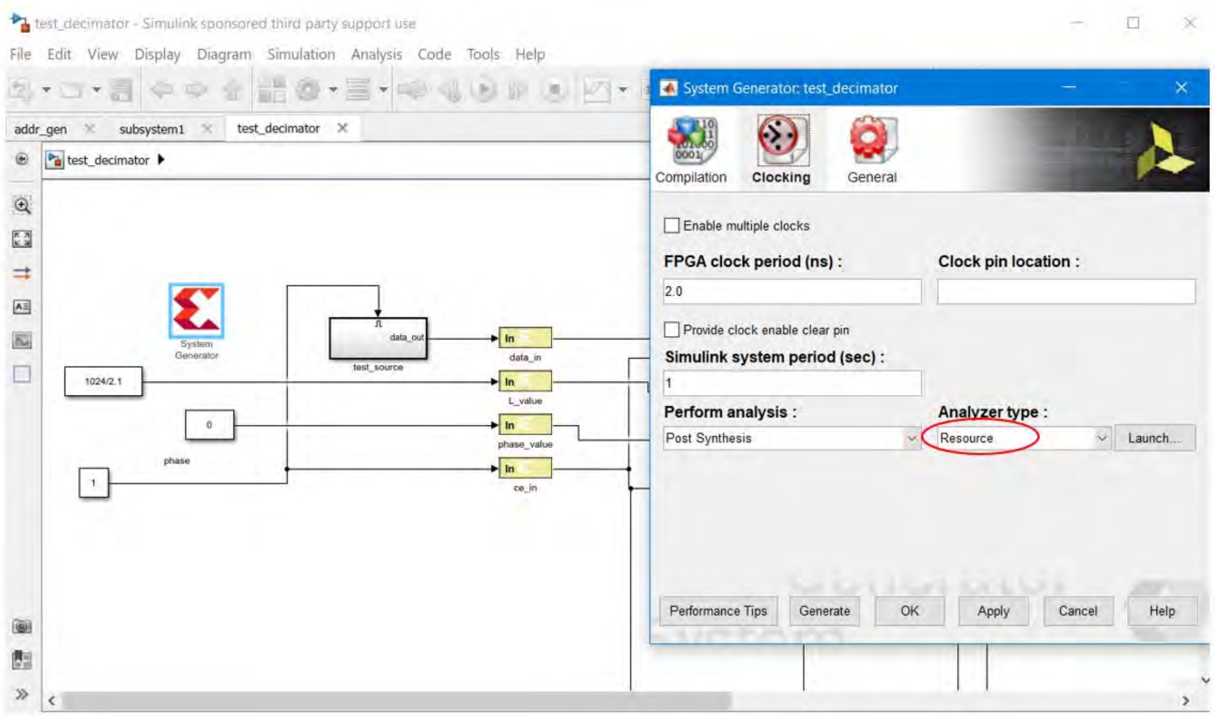
      - Select one of the **Compilation** targets.

        Model Composer can perform resource analysis for any **Compilation** target you select.

      - Specify a **Target Directory**.

   b. In the **Clocking** tab:
      - Set the **Perform Analysis** field to **Post Synthesis** or **Post Implementation** based on the runtime vs. accuracy tradeoff.
      - Set the **Analyzer type** field to **Resource**.

Figure 106: **Resource Analyzer**


test_decimator - Simulink sponsored third party support use
File Edit View Display Diagram Simulation Analysis Code Tools Help
addr_gen subsystem1 test_decimator
System Generator: test_decimator
Compilation Clocking General

Enable multiple clocks: [ ]
FPGA clock period (ns) : 2.0
Clock pin location :
Provide clock enable clear pin: [ ]
Simulink system period (sec) : 1
Perform analysis : Post Synthesis
Analyzer type : Resource

Launch...
Performance Tips Generate OK Apply Cancel Help
System Generator
1024/2.1
0
1
phase
data_out
test_source
In data_in
In L_value
In phase_value
In ce_in

3. In the System Generator token dialog box, click **Generate**.

   When you generate, the following occurs:

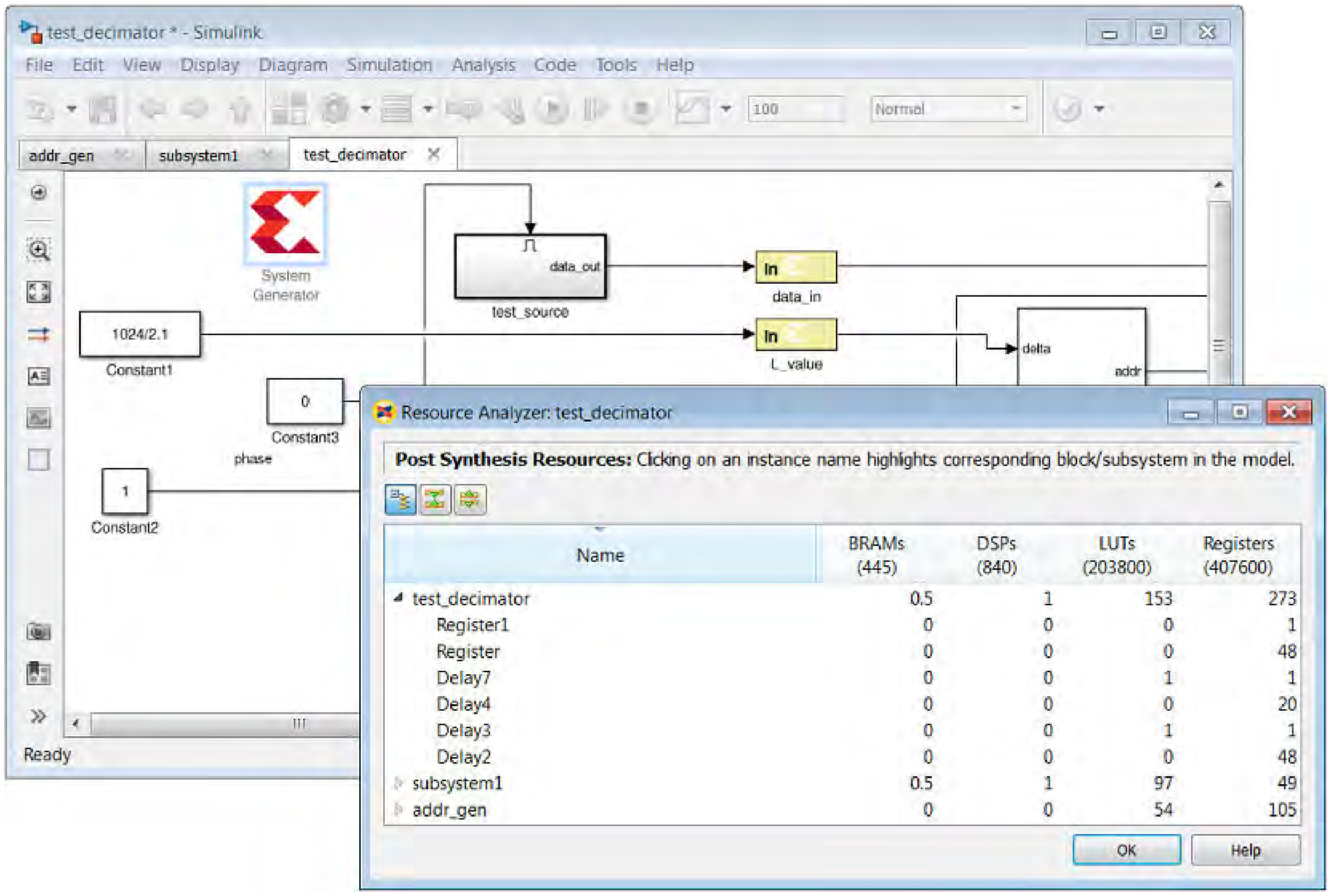
   a. Model Composer generates the required files for the selected compilation target. For resource analysis Model Composer invokes Vivado in the background for the design project.

   b. Depending on your selection for **Perform analysis** (**Post Synthesis** or **Post Implementation**), the design runs in Vivado through synthesis or through implementation.

   c. After the Vivado tools run is completed, resource utilization data is collected from the Vivado resource utilization database and saved in a specific file format under the target directory. At the end of the resource utilization data collection the Vivado project is closed and control is passed to the MATLAB/Model Composer process.

   d. Model Composer processes the resource utilization data and displays a Resource Analyzer table with resource utilization information (see below).

Figure 107: **Resource Analyzer**



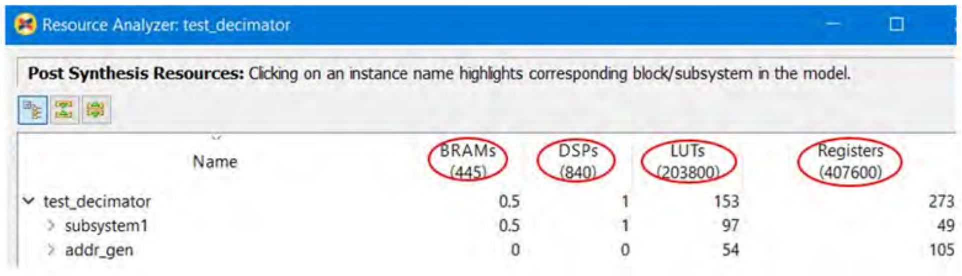
Post Synthesis Resources: Clicking on an instance name highlights corresponding block/subsystem in the model.

Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
test_decimator	0.5	1	153	273
Register1	0	0	0	1
Register	0	0	0	48
Delay7	0	0	1	1
Delay4	0	0	0	20
Delay3	0	0	1	1
Delay2	0	0	0	48
subsystem1	0.5	1	97	49
addr_gen	0	0	54	105

In the resource analyzer table:

- The header section of the dialog box indicates the Vivado design stage after which resource utilization data was collected from Vivado. This will be either **Post Synthesis** or **Post Implementation**.
- The local toolbar contains the following commands to change the display of resource counts:
  - Hierarchical/Flat Display: Toggles the display between a hierarchical tree and a flattened list.
  - Collapse All: Collapses the design hierarchy to display only the top-level objects.
  - Expand All: Expands the design hierarchy at all levels to display resources used by each subsystem and each block in the design.
- The number shown in each column heading indicates the total number of each type of resource available in the Xilinx device for which you are targeting your design. In the example below, the design is targeting a Kintex-7 FPGA.

Figure 108: **Resource Analysis Report for Kintex-7**



Resource Analyzer: test_decimator

Post Synthesis Resources: Clicking on an instance name highlights corresponding block/subsystem in the model.

Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
test_decimator	0.5	1	153	273
subsystem1	0.5	1	97	49
addr_gen	0	0	54	105

- The example displays a hierarchical listing of each subsystem and block in the design, with the count of the following resource types:
  - **BRAMs:** block RAM and FIFO primitives.block RAMs (BRAMs) are counted in this way.

    Table 5: **Number of BRAMs**

Primitive Type	# BRAMs
RAMB36E	1
FIFO36E	1
RAMB18E	0.5
FIFO18E	0.5

    Variations of Primitives (for example, RAM36E1 and RAM36E2) are all counted in the same way.

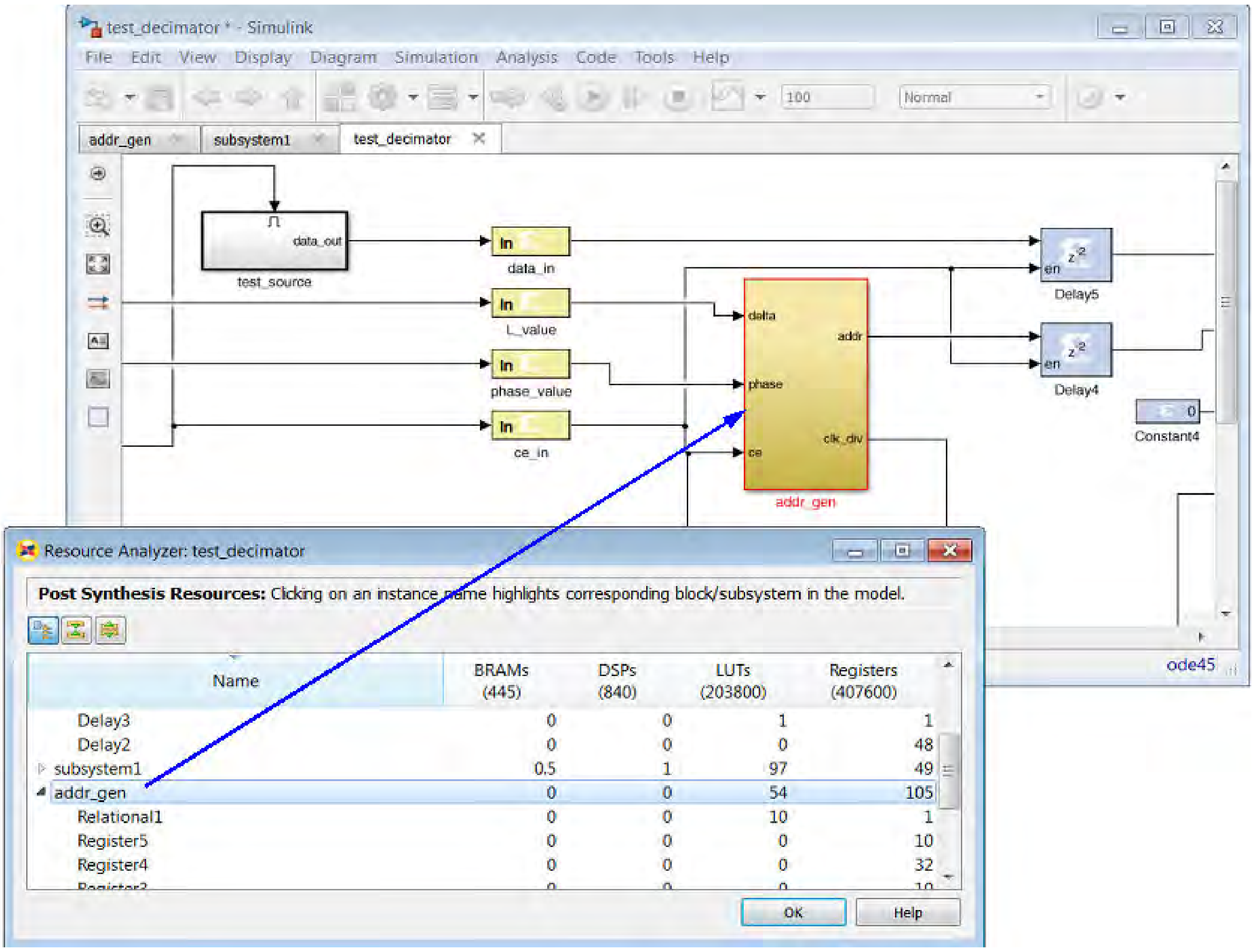
    $$\text{Total BRAMs} = (\text{Number of RAMB36E}) + (\text{Number of FIFO36E}) + 0.5\ (\text{Number of RAMB18E} + \text{Number of FIFO18E})$$

  - **DSPs:** DSP48 primitives (DSP48E, DSP48E1, DSP48E2) and DSP58
  - **Registers:** Registers and Flip-Flops. All primitive names that start with FD* (FDCE, FDPE, FDRE, FDSE, etc.) and LD* (LDCE, LDPE, etc.) are considered as **Registers**.
  - **LUTs:** All LUT types combined.
- The display order can be sorted for any column's values by clicking the column head.
- You can cross probe from the table to the Simulink model by selecting a row in the table, which will highlight the corresponding HDL blocks in the Simulink model. See Cross Probing from the Resource Analysis Results to the Model.

Cross Probing from the Resource Analysis Results to the Model

You can cross probe from the Resource Analyzer table to the Simulink® model by clicking a block or subsystem name in the Resource Analyzer table, which highlights the corresponding HDL block or subsystem in the model. The cross probing is useful to identify blocks and subsystems that are implemented using a particular type of resource.

Figure 109: **Resource Analyzer**



Resource Analyzer: test_decimator

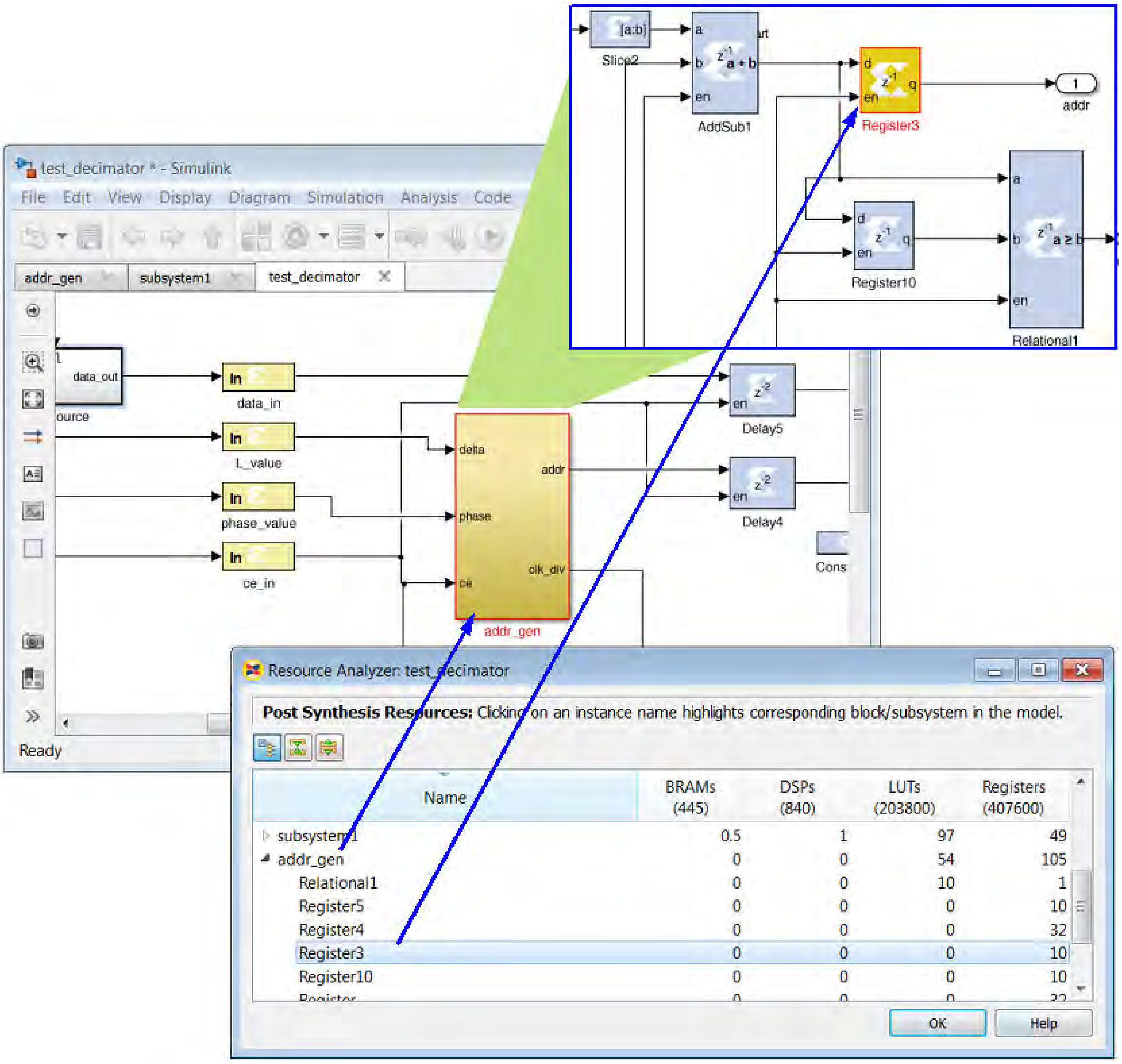
Post Synthesis Resources: Clicking on an instance name highlights corresponding block/subsystem in the model.

Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
Delay3	0	0	1	1
Delay2	0	0	0	48
subsystem1	0.5	1	97	49
addr_gen	0	0	54	105
Relational1	0	0	10	1
Register5	0	0	0	10
Register4	0	0	0	32

When you cross probe, the following will display in the model:

- The block you have selected in the table will be highlighted in yellow and outlined in red.
- If the block or subsystem you have selected in the table is within an upper-level subsystem, then the parent subsystem is highlighted in red in addition to the underlying block.

Figure 110: **Selected Subsystem**



Post Synthesis Resources: Clicking on an instance name highlights corresponding block/subsystem in the model.

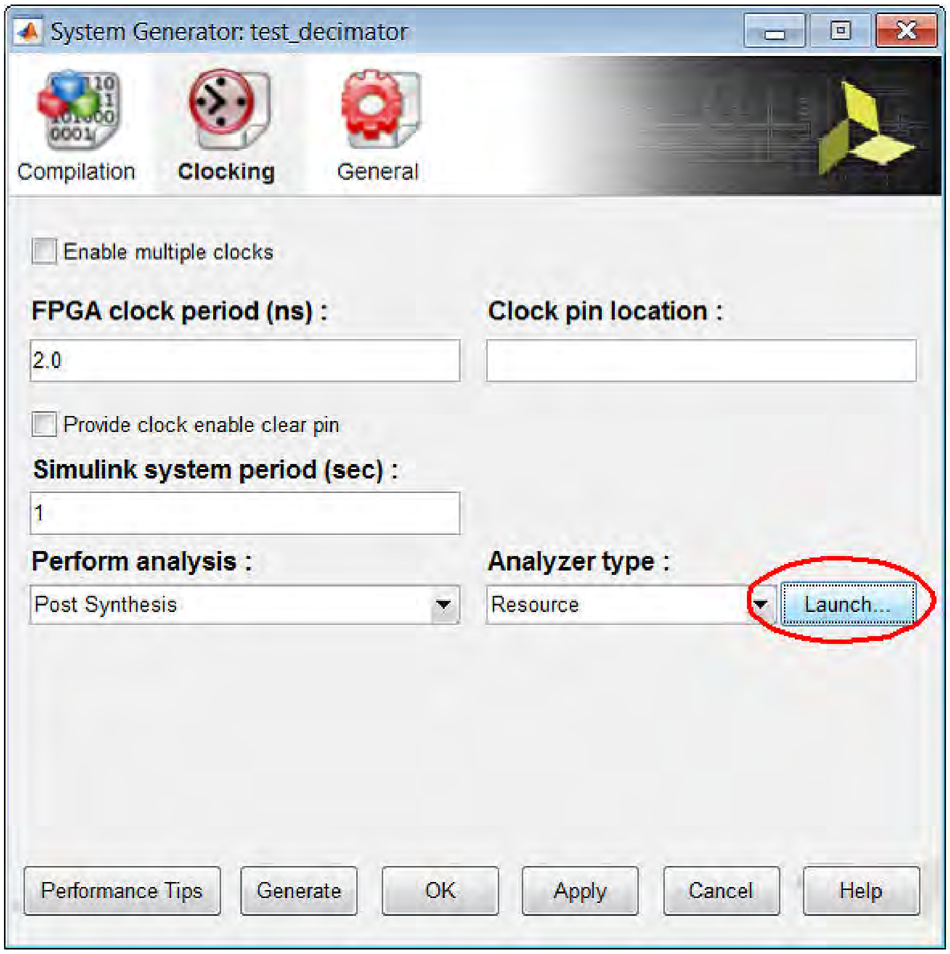
Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
▷ subsystem1	0.5	1	97	49
◢ addr_gen	0	0	54	105
Relational1	0	0	10	1
Register5	0	0	0	10
Register4	0	0	0	32
Register3	0	0	0	10
Register10	0	0	0	10

Accessing Existing Resource Analysis Results

A **Launch** button is provided under the **Clocking** tab of the System Generator token dialog box to launch the Resource Analyzer table using the existing resource utilization results for the model. Make sure the **Target directory** specified on the **Compilation** tab of the dialog box is readable by the Resource Analyzer, and the **Analyzer type** field is set to **Resource**. This will only work if you already ran analysis on the Simulink model and haven't changed the Simulink model since the last run.

When you click the **Launch** button, the Resource Analyzer table will display the resource utilization results stored in the **Target directory** specified on the **Compilation** tab, regardless of the option selected for **Perform analysis** (the **Post Synthesis** or **Post Implementation** option).

Figure 111: **Launch Button**


System Generator: test_decimator

Enable multiple clocks: ☐

FPGA clock period (ns) : 2.0

Clock pin location :

Provide clock enable clear pin: ☐

Simulink system period (sec) : 1

Perform analysis : Post Synthesis

Analyzer type : Resource

You can launch the Resource Analyzer table to display existing resource utilization results by entering the following command at the MATLAB® command prompt:

```
>> xlAnalyzeResource(get_param('model_name','handle'),'./netlist')
```

- `get_param('model_name','handle')` gives you the model handle.

  Note: `model_name` represents the name of the model.

- For the path to `netlist` directory, you can use an absolute path, or if you are using this API from the same directory where `netlist` directory is present, then you can use a relative path like '`./netlist`'.

Recommendations for Optimizing Resource Analysis

The following are recommended for using the Resource Analyzer to optimize resource utilization in the design:

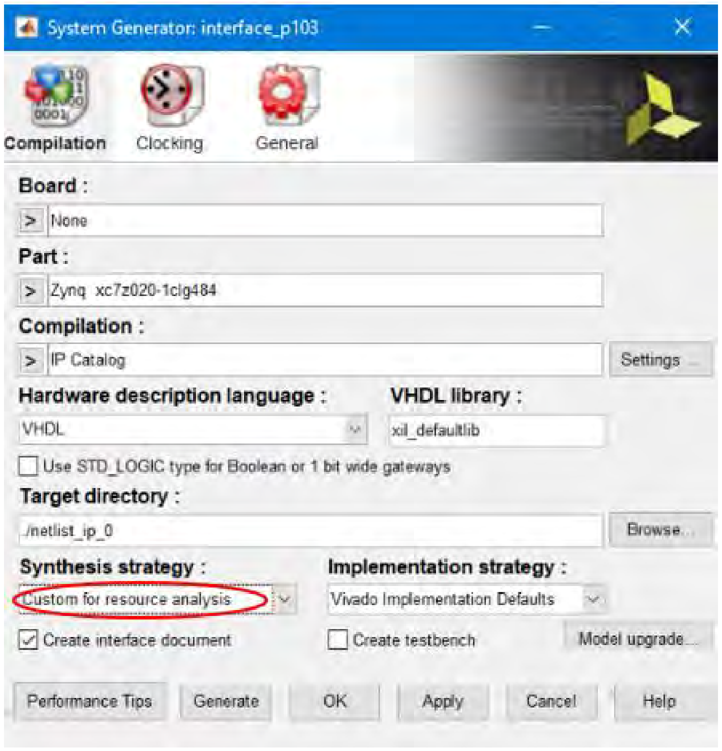
- For quicker resource analysis iterations, post-synthesis analysis is preferred over post-implementation analysis.
- After logic optimization during the Vivado Synthesis process the tool does not keep information about merged logic in the Vivado database. Merged and shared logic may make it difficult to accurately cross probe from Vivado resource data to the Simulink model. Hence, it is recommended that you create a custom Vivado Synthesis strategy to control merged and shared logic.

  For information about how to create a custom Synthesis strategy in Vivado IDE, see this link in the *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893).

  To control merged and shared logic in the Vivado IDE, make the following changes to the default Vivado Synthesis strategy.

  1. In Vivado IDE:
     - Select the Synthesis option `-keep_equivalent_registers`.
     - Set the Synthesis option `-resource_sharing` to the value `off`.
  2. Save the new Synthesis strategy and exit Vivado IDE.
  3. In Model Composer, select the new custom **Synthesis strategy** in the System Generator token dialog box before generating the design.

Figure 112: **Synthesis Strategy**



Board : None

Part : Zynq xc7z020-1clg484

Compilation : IP Catalog

Hardware description language : VHDL

VHDL library : xil_defaultlib

☐ Use STD_LOGIC type for Boolean or 1 bit wide gateways

Target directory : ./netlist_ip_0

Synthesis strategy : Custom for resource analysis

Implementation strategy : Vivado Implementation Defaults

☑ Create interface document

☐ Create testbench

Using Hardware Co-Simulation

Model Composer provides hardware co-simulation, making it possible to incorporate a design running in an FPGA directly into a Simulink® simulation. This allows all (or a portion) of the Model Composer design that had been simulating in Simulink as sequential software to be executed in parallel on the FPGA, and can speed up simulation dramatically. Users of this flow can send larger data sets, or more test vectors, doing an exhaustive functional test of the implemented logic. This increased code coverage allows more corner cases to be verified to help identify design bugs in the logic. Data at the input to the compiled co-simulation block on the Simulink model is sent to the target FPGA, either as one transaction or a burst of transactions, executed for a given number of clock cycles in parallel, and read back to the model's co-simulation outputs.

Hardware co-simulation has two compilation types: burst or non-burst (standard). The burst mode provides much higher performance. Channels to each input of the compiled co-simulation target are opened and packets of data are sent to the open channel, followed by bursting to all of the remaining inputs. The FPGA design is executed in parallel for enough cycles to consume the data, and the target outputs are burst read in a channelized fashion. Bursting provides for less overhead to send and receive large amounts of data from the FPGA. However, burst mode is only supported through MATLAB® script-based hardware co-simulation of the Hardware Co-Simulation target and is not used within Simulink. Exhaustive data vectors can be scripted to test the functionality of the co-simulation target, and an example script is returned as part of the compilation. Non-burst mode has lower performance but allows a compiled co-simulation block to be used within Simulink in place of the original Model Composer design hierarchy.

***Note*:** Hardware co-simulation does not support designs which contain multiple clocks.

Board support allows the JTAG-based physical interface to communicate with the co-simulation target: JTAG-based communication is available for most of the JTAG aware boards that exist as a project target in Vivado®. Boards from Xilinx partners are available and can be downloaded from the partner websites and installed as part of the Vivado Design Suite. Custom boards can also be created as detailed in Appendix A, Board Interface File, in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895). Setting up board awareness in Model Composer and the minimum tags needed in the `board.xml` file are detailed in the section Specifying Board Support in Model Composer HDL Blockset.

Hardware Co-Simulation compilation targets automatically create a bitstream based on the selected communication interface and associate it to a block.

- If a board is supported for JTAG hardware co-simulation, the **Hardware Co-Simulation** option for **Compilation** is enabled in the System Generator token dialog box when you perform the procedure described in Compiling a Model for Hardware Co-Simulation. If the **Hardware Co-Simulation** option is grayed out and disabled, you cannot perform JTAG hardware co-simulation on the board.

  This support applies to the following types of boards:

  Table 6: **Board Support**

Board Name	Display Name
zed	ZedBoard Zynq Evaluation and Development Kit
ac701	Artix-7 AC701 Evaluation Platform 1.0/1.1
kc705	Kintex-7 KC705 Evaluation Platform 1.0/1.1
kcu105	Kintex-UltraScale KCU105 Evaluation Platform
vc707	Virtex-7 VC707 Evaluation Platform
vc709	Virtex-7 VC709 Evaluation Platform
vcu108	Virtex-UltraScale VCU108 Evaluation Platform
zc702	ZYNQ-7 ZC702 Evaluation Board
zc706	ZYNQ-7 ZC706 Evaluation Board

  - Xilinx boards installed as part of your Vivado installation.
  - Partner boards, which are available and can be downloaded from the partner websites and installed as part of the Vivado Design Suite,

- Custom boards, which can be created in the Vivado Design Suite as detailed in Appendix A, Board Interface File, in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895).

Compiling a Model for Hardware Co-Simulation

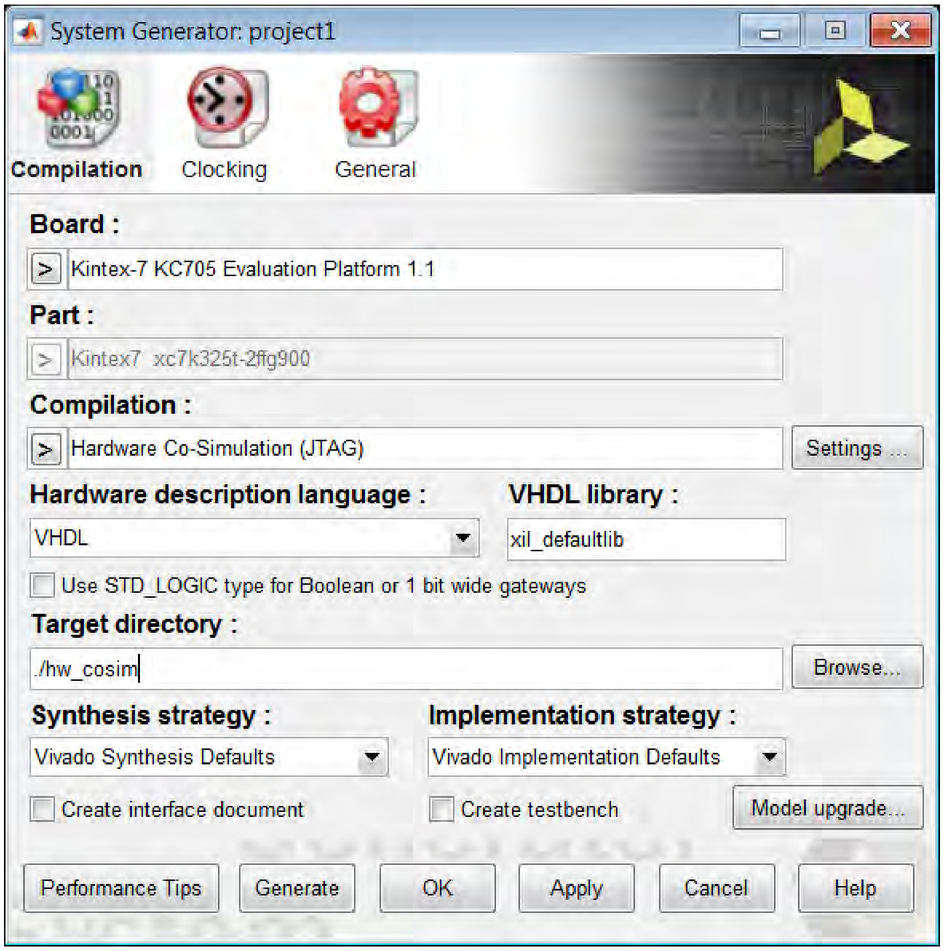
The starting point for hardware co-simulation is the Model Composer model or subsystem you would like to run in hardware. A model can be co-simulated if it meets the requirements of the underlying hardware board. The model must include a System Generator token; this block defines how the model should be compiled into hardware.

For information on how to use the System Generator token, see Compiling and Simulating Using the System Generator Token.

To compile your Model Composer model for hardware co-simulation, perform the following:

1. Double-click the System Generator token to open the System Generator token dialog box.

Figure 113: **System Generator Token Dialog Box**


System Generator: project1

Compilation | Clocking | General

Board : Kintex-7 KC705 Evaluation Platform 1.1

Part : Kintex7 xc7k325t-2ffg900

Compilation : Hardware Co-Simulation (JTAG)

Hardware description language : VHDL

VHDL library : xil_defaultlib

Use STD_LOGIC type for Boolean or 1 bit wide gateways ☐

Target directory : ./hw_cosim

Synthesis strategy : Vivado Synthesis Defaults

Implementation strategy : Vivado Implementation Defaults

Create interface document ☐

Create testbench ☐

2. In the **Compilation** tab, select a **Board** and a version of the board.

   The boards appearing in the **Board** list are:

   - All of the boards installed as part of the Vivado.
   - Any custom boards you have created in the Vivado.
   - Any Partner boards you have purchased and enabled in the Vivado.

   For a Partner board or a custom board to appear in the **Board** list, you must configure Model Composer to access the board files that describe the board. Board awareness in Model Composer is detailed in Specifying Board Support in Model Composer HDL Blockset.

   To compile for hardware co-simulation, you must select a **Board**. You cannot set the **Board** field to **None** and select a **Part** instead of a **Board**.

   When you select a **Board**, the **Part** field displays the name of the Xilinx device on the selected **Board**, and the **Part** setting cannot be changed.

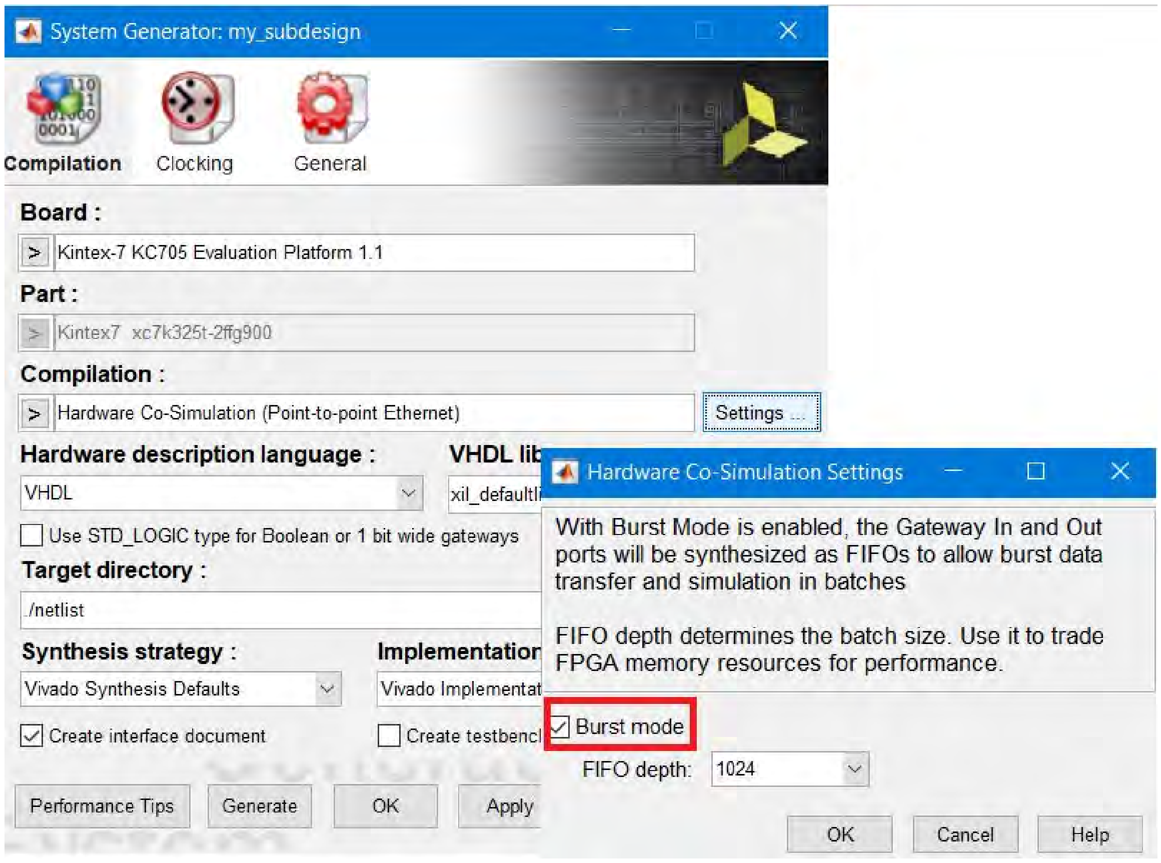
3. In the **Compilation** field, select **Hardware Co-Simulation** and, further select **JTAG** interface to perform hardware co-simulation.

   If the **Hardware Co-Simulation** option is grayed out and disabled, you cannot perform JTAG hardware co-simulation on the selected board.

4. *If you will use burst mode* for a faster hardware co-simulation run, click the **Settings** button next to the **Compilation** field, select **Burst mode**, and enter a **FIFO depth** for the burst mode operation. Then click **OK** to close the Hardware Co-Simulation Settings dialog box.

   For a description of the burst mode, see Burst Data Transfers for Hardware Co-Simulation.

Figure 114: **Burst Mode**


System Generator: my_subdesign

Compilation | Clocking | General

Board : Kintex-7 KC705 Evaluation Platform 1.1

Part : Kintex7 xc7k325t-2ffg900

Compilation : Hardware Co-Simulation (Point-to-point Ethernet) | Settings ...

Hardware description language : VHDL

VHDL lib: xil_defaultli

☐ Use STD_LOGIC type for Boolean or 1 bit wide gateways

Target directory : ./netlist

Synthesis strategy : Vivado Synthesis Defaults

Implementation: Vivado Implementat

☑ Create interface document

☐ Create testbench

Performance Tips | Generate | OK | Apply

Hardware Co-Simulation Settings

With Burst Mode is enabled, the Gateway In and Out ports will be synthesized as FIFOs to allow burst data transfer and simulation in batches

FIFO depth determines the batch size. Use it to trade FPGA memory resources for performance.

☑ Burst mode

FIFO depth: 1024

OK | Cancel | Help

IMPORTANT! *To perform a burst mode hardware co-simulation, you must create a test bench by checking the* ***Create Testbench*** *box in the System Generator token dialog box.*

5. If you want to create a test bench as part of the compilation, select the **Create Testbench** option.

   If you select **Create Testbench**, the compilation will automatically create an example test bench for you. You can also create your own test bench for hardware co-simulation (see M-Code Access to Hardware Co-Simulation).

6. Click the **Generate** button.

   The code generator produces an FPGA configuration bitstream for your design that is suitable for hardware co-simulation. Model Composer not only generates the HDL and netlist files for your model during the compilation process, but it also runs the downstream tools necessary to produce an FPGA configuration file.

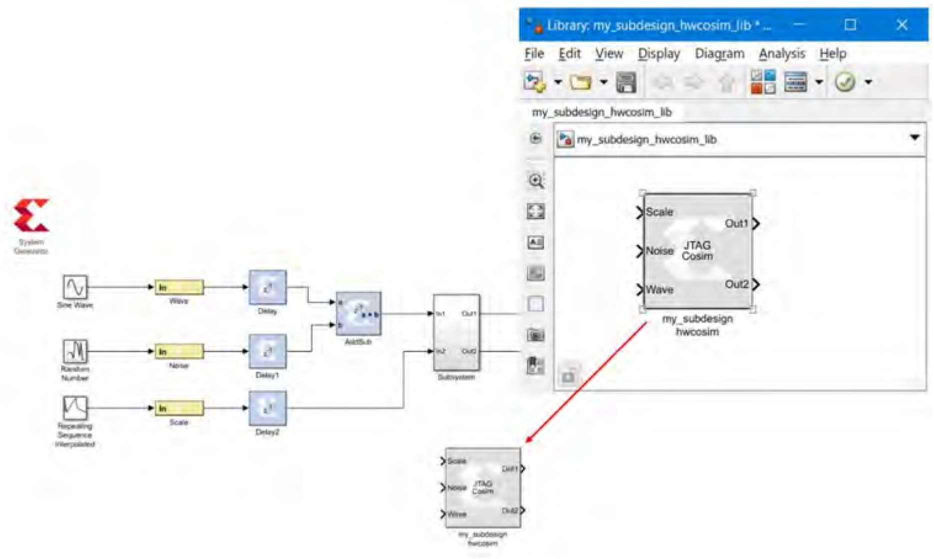
The configuration bitstream contains the hardware associated with your model, and also contains additional interfacing logic that allows Model Composer to communicate with your design using a physical interface between the board and the PC. This logic includes a memory map interface over which Model Composer can read and write values to the input and output ports on your design. It also includes any board-specific circuitry that is required for the target FPGA board to function correctly.

When the Compilation finishes the results are as follows:

- If you *have not* selected **Burst mode** in step 4 above (standard mode), a **JTAG Cosim** hardware co-simulation block will appear in a separate window. Drag (or Copy and Paste) the Hardware Cosim block into your Simulink model. The Hardware Cosim block will enable you to perform hardware co-simulation from within the Simulink window.

  For a description of the hardware co-simulation block, see Hardware Co-Simulation Blocks.

  Figure 115: **Hardware Co-Simulation Library Block**

  If you selected the **Create Testbench** option for compilation, an M-Code HWCosim example test bench will also be generated (see M-Code Access to Hardware Co-Simulation) by the compilation. You can use this test bench to perform hardware co-simulation, or customize this test bench to develop a test bench of your own.

- If you *have* selected **Burst mode** in step 4 above (burst mode), no hardware co-simulation block will appear. When you perform the burst mode co-simulation, you will use the MATLAB® M-code test bench placed in the target directory during compilation.

  - If you compiled the top-level design the test bench will be named:

    ```
    <design_name>_hwcosim_test.m
    ```

- If you compiled a subsystem of the design the test bench will be named:

  ```
  <design_name>_<sub_system>_hwcosim_test.m
  ```

The compilation has prepared the Simulink model for performing hardware co-simulation.

To perform the hardware co-simulation, proceed as follows:

- To perform the standard (non-burst mode) hardware co-simulation, see Performing Standard Hardware Co-Simulation.
- To perform the burst mode hardware co-simulation, see Performing Burst Mode Hardware Co-Simulation.

Performing Standard Hardware Co-Simulation

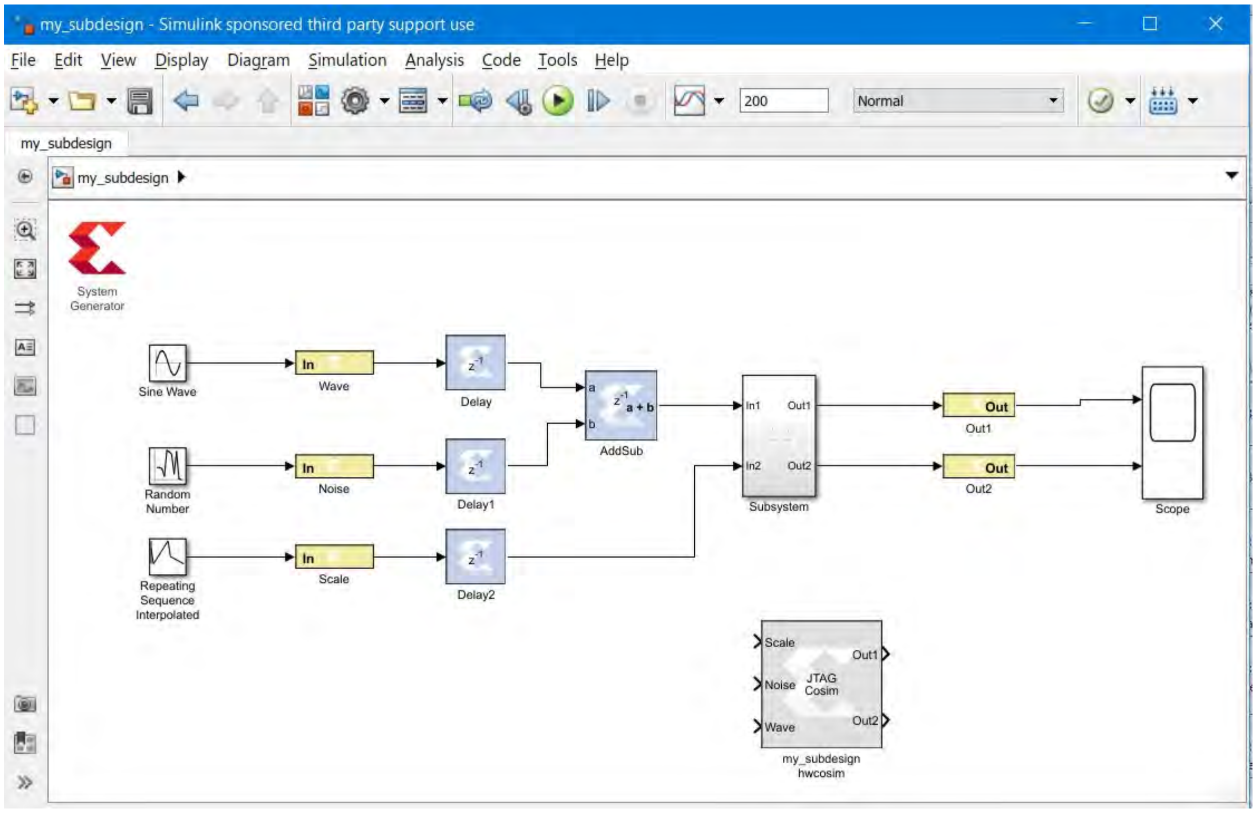
If you are performing the standard (non-burst mode) hardware co-simulation, your Simulink model will contain a JTAG hardware co-simulation block. This block was created automatically when Model Composer finished compiling your design into an FPGA bitstream (see Compiling a Model for Hardware Co-Simulation). The block is stored in a Simulink library with this file name:

```
<design_name>_hwcosim_lib.slx
```

The hardware co-simulation block was moved into your Simulink model at the end of the compilation procedure. In the following procedure, you will have to wire up this block in your Simulink model to perform hardware co-simulation.

***Note*:** If your board contains a Zynq® SoC device, you must install the Vitis™ unified software platform with the Vivado® Design Suite to perform hardware co-simulation.

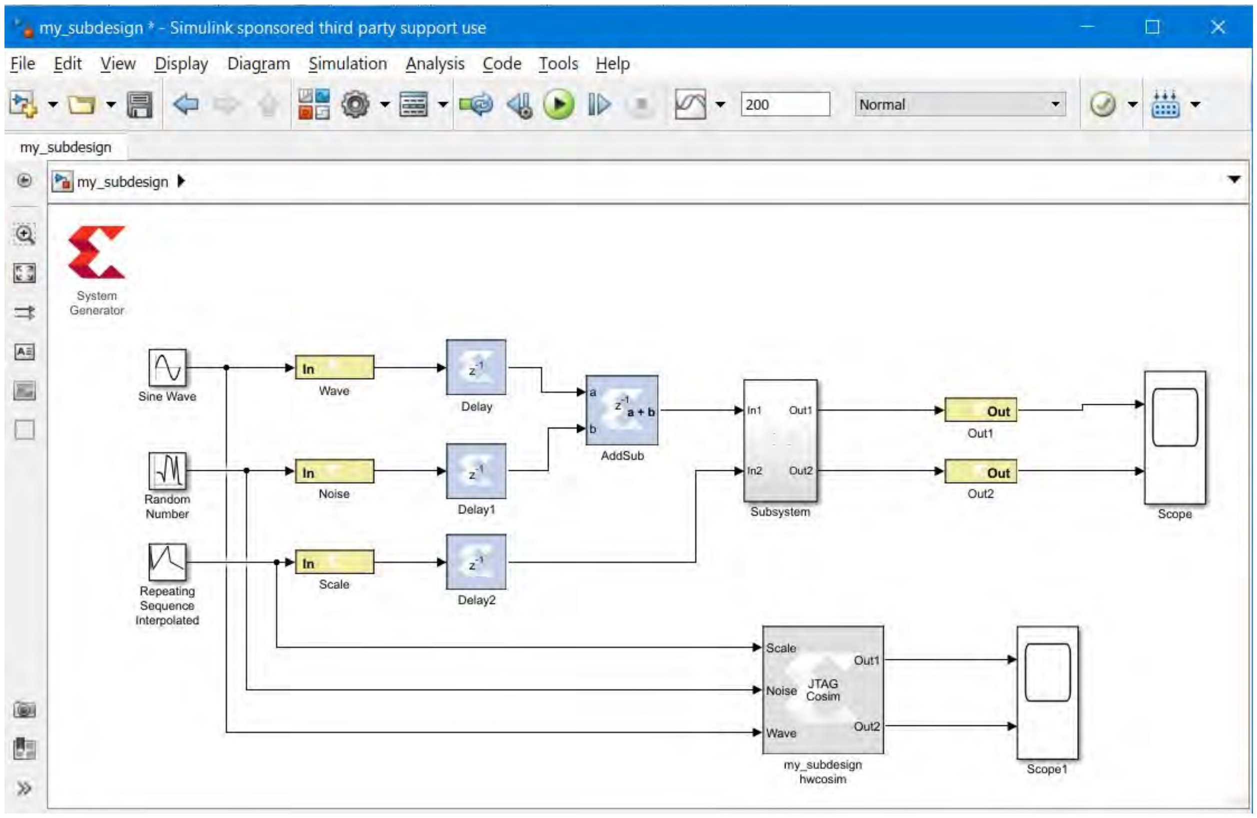
Figure 116: **Hardware Co-Simulation Block**



To perform the standard hardware co-simulation:

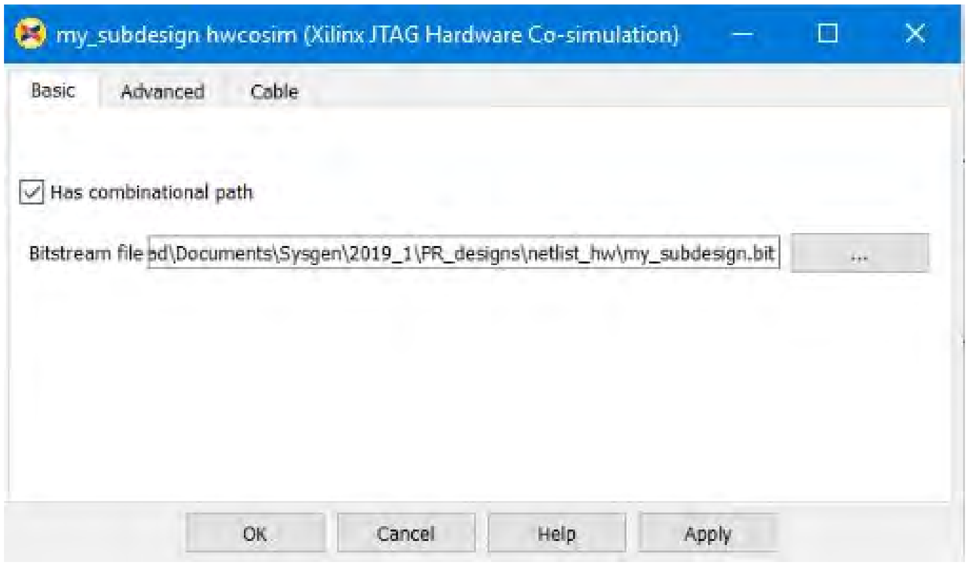
1. Connect the hardware co-simulation block to the Simulink blocks that supply its inputs and receive its outputs.

Figure 117: **Connect to Simulink Blocks**



2. Double-click the hardware co-simulation block to display the properties dialog box for the block.

Figure 118: **Hardware Co-Simulation Library Block Properties**



☑ Has combinational path

Bitstream file: ad\Documents\Sysgen\2019_1\PR_designs\netlist_hw\my_subdesign.bit

3. Fill out the block parameters in the properties dialog box.

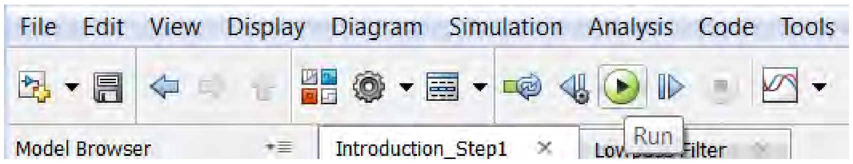
The properties are described in Block Parameters for the JTAG Hardware Co-Simulation Block .

4. To set up the board for performing JTAG hardware co-simulation.You should connect a cable to the board's JTAG port.

   For a description of the setup procedure for a JTAG hardware co-simulation, using a KC705 board as an example, see Setting Up a KC705 Board for JTAG Hardware Co-Simulation.

5. In the Simulink model, simulate the model and the hardware by selecting **Simulation → Run** or clicking the **Run** simulation button.

Figure 119: **Run Button**



   Running the simulation will simulate both the Model Composer design (or subsystem) in your Simulink model and the Xilinx device on your target board. You can then examine the results of the two simulations and compare the results to determine if the design implemented in hardware will operate as expected.

Performing Burst Mode Hardware Co-Simulation

To perform the burst mode hardware co-simulation, you will execute the MATLAB M-code test bench that was generated automatically during compilation (see Compiling a Model for Hardware Co-Simulation).

This test bench resides in the **Target directory** specified when the design was compiled for the hardware co-simulation compilation target.

The test bench is named as follows:

- If you compiled the top-level design the test bench will be named:

```
<design_name>_hwcosim_test.m
```

- If you compiled a subsystem of the design the test bench will be named:

```
<design_name>_<sub_system>_hwcosim_test.m
```

Note: If your board contains a Zynq® SoC device, you must install the Vitis™ unified software platform with the Vivado to perform hardware co-simulation.

To perform burst mode hardware co-simulation,

1. Set up the board for performing JTAG hardware co-simulation.
   - Connect a cable to the board's JTAG port.

For a description of the setup procedure for a JTAG hardware co-simulation, using a KC705 board as an example, see Setting Up a KC705 Board for JTAG Hardware Co-Simulation.

2. Run the test bench script from the MATLAB console. To run the test bench script, you can open the MATLAB console, change directory to the **Target directory** and run the script by name.

   The script runs the Simulink model to determine the stimulus data driven to the Xilinx Gateway In blocks (from the other Simulink source blocks or MATLAB variables), and captures the expected output produced by the Xilinx block design (BD), and exports the data to the **Target directory** as these separate data files:

   ```
   <design_name>_<sub_system>_<port_name>.dat
   ```

   The test bench then compares actual to expected outputs.

   If the test fails this will be printed on the console, and the failing comparisons will be listed in this file:

   ```
   <design_name>_<sub_system>_hwcosim_test.result
   ```

M-Code Access to Hardware Co-Simulation

It is possible to programmatically control the hardware created through the Model Composer HDL hardware co-simulation flow using MATLAB M-code (M-Hwcosim). The M-Hwcosim interfaces allow for MATLAB objects that correspond to the hardware to be created in pure M-code, independent of the Simulink framework. These objects can then be used to read and write data into hardware. This capability is useful for providing a scripting interface to hardware co-simulation, allowing for the hardware to be used in a scripted test bench or deployed as hardware acceleration in M-code.

For more information on this subject, refer to M-Code Access to Hardware Co-Simulation.

Setting Up Your Hardware Board

The first step in performing hardware co-simulation is to set up your hardware board. The hardware setup for JTAG hardware co-simulation is as follows:

For JTAG-based hardware co-simulation, you will connect a cable to the board's JTAG port. Consult your board's documentation for the location of the board's JTAG port. Documentation for Xilinx boards can be downloaded from the Boards and Kits page on the Xilinx website.

For a description of the setup procedure for a JTAG hardware co-simulation, using a KC705 board as an example, see Setting Up a KC705 Board for JTAG Hardware Co-Simulation

Setting Up a KC705 Board for JTAG Hardware Co-Simulation

The following procedure describes how to set up the hardware required to run JTAG hardware co-simulation on a KC705 board.

For detailed information about the KC705 board, see the *KC705 Evaluation Board for the Kintex-7 FPGA User Guide* (UG810).

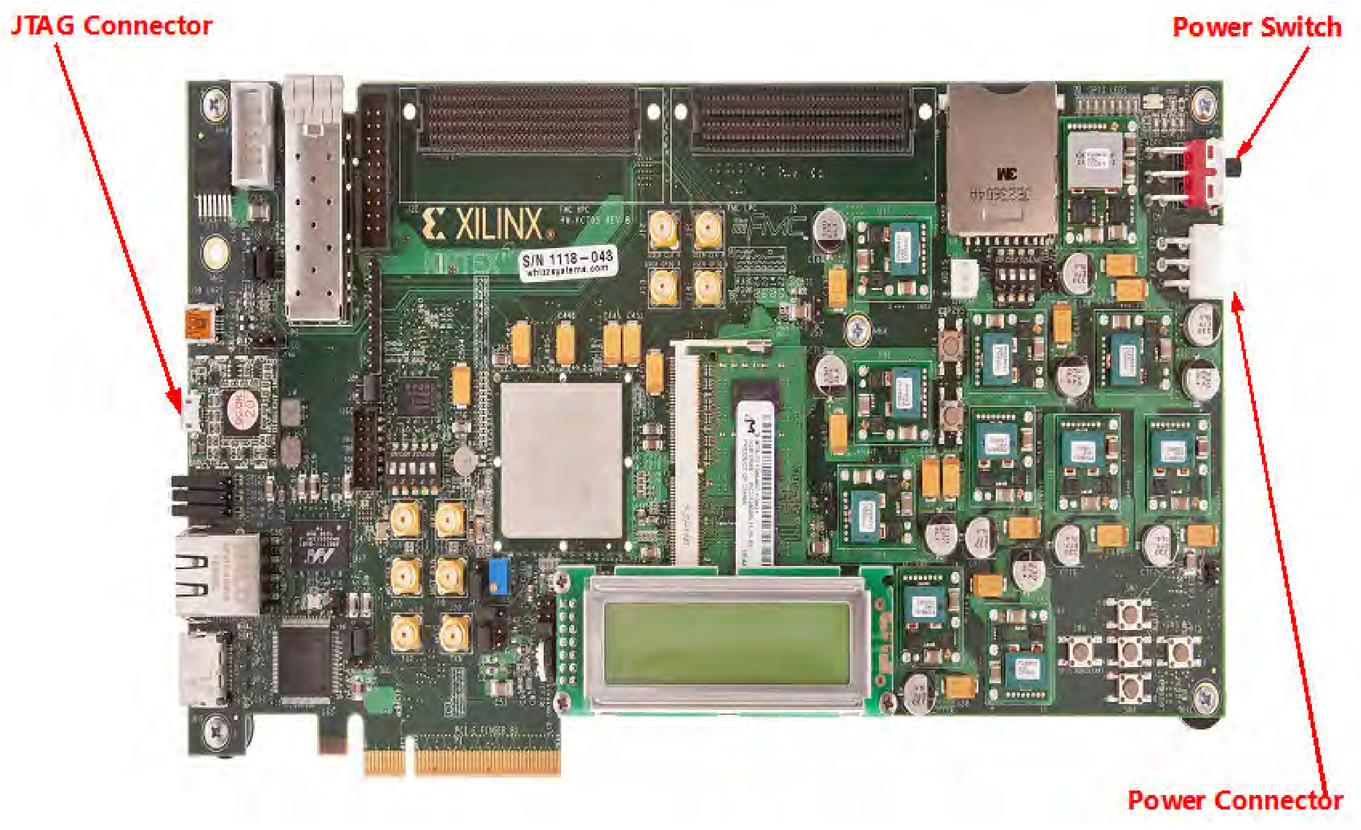
Assemble the Required Hardware

1. Xilinx Kintex-7 KC705 board which includes the following:
   a. Kintex-7 KC705 board
   b. 12V Power Supply bundled with the KC705 kit
   c. Micro USB-JTAG cable

Set Up the KC705 Board

The figure below illustrates the KC705 components of interest in this JTAG setup procedure:

Figure 120: **KC705 Board**



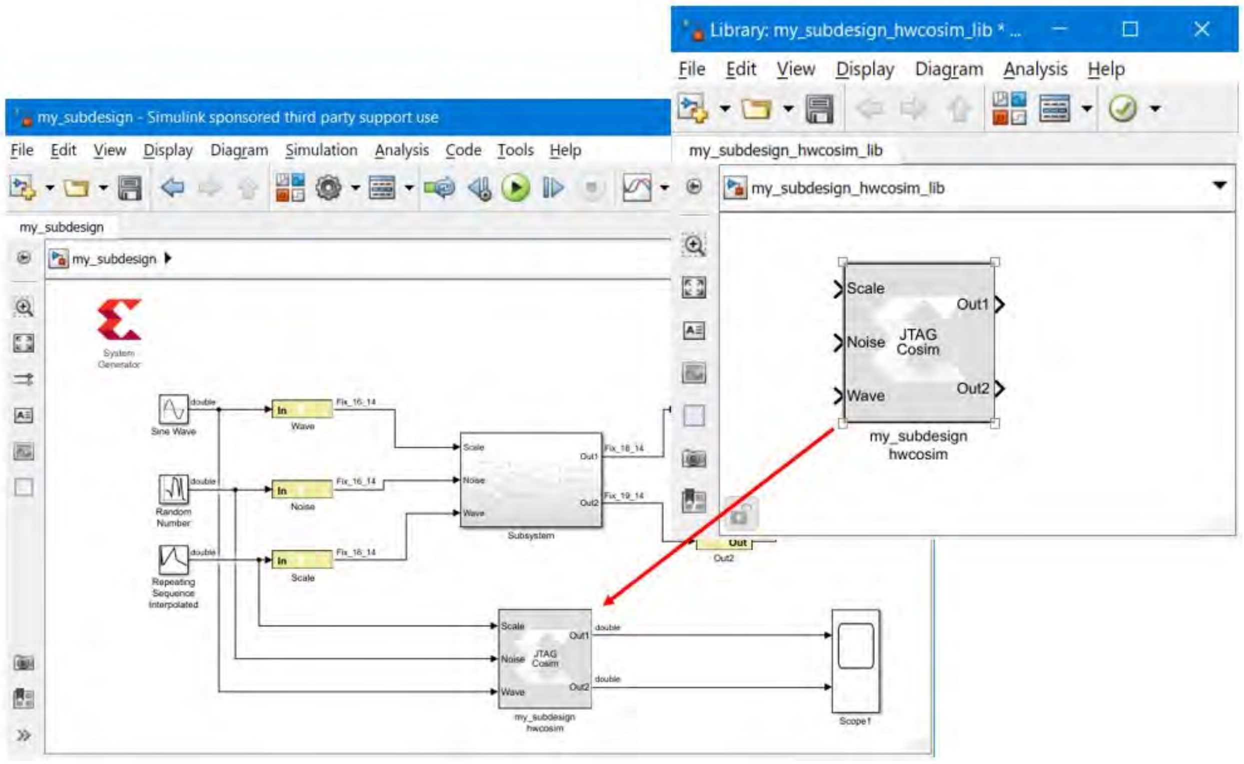
1. Position the KC705 board as shown above.

2. Make sure the power switch, located in the upper-right corner of the board, is in the OFF position.
3. Connect the AC power cord to the power supply brick. Plug the power supply adapter cable into the KC705 board. Plug in the power supply to AC power.
4. Connect the small end of the Micro USB-JTAG cable to the JTAG socket.
5. Connect the large end of the Micro USB-JTAG cable to a USB socket on your PC.
6. Turn the KC705 board Power switch ON.

Hardware Co-Simulation Blocks

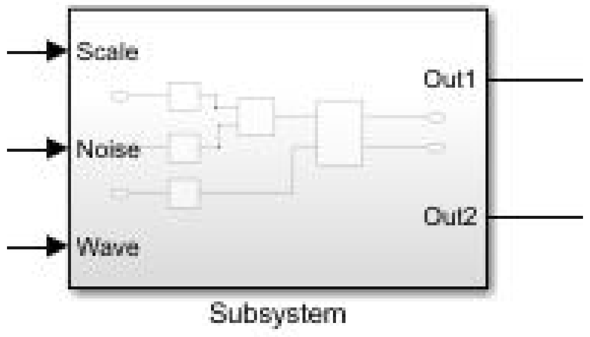
Model Composer automatically creates a new hardware co-simulation block once it has finished compiling your design into an FPGA bitstream. It also creates a Simulink library to store the hardware co-simulation block. At this point, you can copy the block out of the library and use it in your Model Composer design like any other Simulink or HDL blocks.

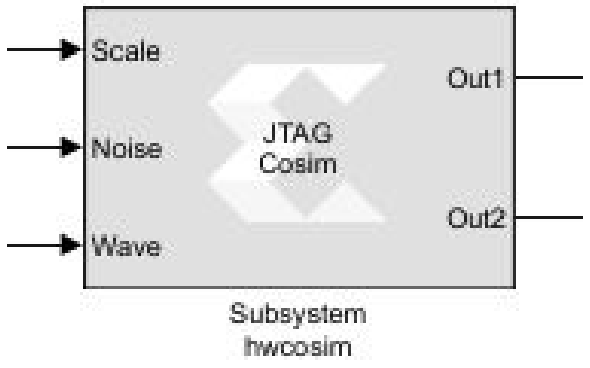
Figure 121: **Hardware Co-Simulation Blocks**


Library: my_subdesign_hwcosim_lib * ...
File Edit View Display Diagram Analysis Help
my_subdesign_hwcosim_lib
Scale
Noise
Wave
Out1
Out2
JTAG Cosim
my_subdesign hwcosim
my_subdesign - Simulink sponsored third party support use
File Edit View Display Diagram Simulation Analysis Code Tools Help
my_subdesign
System Generator
Sine Wave
Random Number
Repeating Sequence Interpolated
In
Wave
Noise
Scale
Subsystem
Scope1

The hardware co-simulation block assumes the external interface of the model or Subsystem from which it is derived. The port names on the hardware co-simulation block match the ports names on the original Subsystem. The port types and rates also match the original design.

Figure 122: **Port Names**





Hardware co-simulation blocks are used in a Simulink design the same way other blocks are used. During simulation, a hardware co-simulation block interacts with the underlying FPGA board, automating tasks such as device configuration, data transfers, and clocking. A hardware co-simulation block consumes and produces the same types of signals that other Model Composer HDL blocks use. When a value is written to one of the block's input ports, the block sends the corresponding data to the appropriate location in hardware. Similarly, the block retrieves data from hardware when there is an event on an output port.

hardware co-simulation blocks may be driven by Xilinx® fixed-point signal types, Simulink fixed-point signal types, or Simulink doubles. Output ports assume a signal type that is appropriate for the block they drive. If an output port connects to an HDL block, the output port produces a Xilinx® fixed-point signal. Alternatively, the port produces a Simulink data type when the port drives a Simulink block directly.

***Note*:** When Simulink data types are used as the block signal type, quantization of the input data is handled by rounding, and overflow is handled by saturation.

Like other HDL blocks, hardware co-simulation blocks provide parameter dialog boxes that allow them to be configured with different settings. The parameters that a hardware co-simulation block provides depend on the FPGA board the block is implemented for (i.e., different FPGA boards provide their own customized hardware co-simulation blocks).

Block Parameters for the JTAG Hardware Co-Simulation Block

The block parameters dialog box for the JTAG hardware co-simulation block can be invoked by double-clicking the block icon in your Simulink model.

Parameters specific to the block are as follows:

Basic tab

Has combinational path: Select this if your circuit has any combinational paths. A combinational path is one in which a change propagates from input to output without any clock event. There is no latch, flip-flop, or register in the path. Enabling this option causes Model Composer to read the outputs immediately after writing inputs, before clocking the design. This ensures that value changes on combinational paths extending from the hardware co-simulation block into the Simulink Model get propagated correctly.

Bitstream file: Specify the FPGA configuration bitstream. By default this field contains the path to the bitstream generated by Model Composer during the last **Generate** triggered from the System Generator Token.

Advanced tab

Skip device configuration: When selected, the configuration bitstream will not be loaded into the FPGA or SoC. This option can be used if another program is configuring the device (for example, the Vivado Hardware Manager and the Vivado Logic Analyzer).

Display Part Information: This option toggles the display of the device part information string (for example, **xc7k325tffg900-2** for a Kintex device) in the center of the hardware co-simulation block.

Cable tab

Cable Settings

- **Type:** Currently, **Auto Detect** is the only setting for this parameter. Model Composer will automatically detect the cable type.

Hardware Co-Simulation Clocking

If you are performing a standard hardware co-simulation, you will have to select a clocking mode when you configure the co-simulation block. included in your Simulink model.

Clocking Modes

There are several ways in which a Model Composer hardware co-simulation block can be synchronized with its associated FPGA hardware. In single-step clock mode, the FPGA is in effect clocked from Simulink, whereas in free-running clock mode, the FPGA runs off an internal clock, and is sampled asynchronously when Simulink wakes up the hardware co-simulation block.

Single-Step Clock

In single-step clock mode, the hardware is kept in lock step with the software simulation. This is achieved by providing a single clock pulse (or some number of clock pulses if the FPGA is over-clocked with respect to the input/output rates) to the hardware for each simulation cycle. In this mode, the hardware co-simulation block is bit-true and cycle-true to the original model.

Because the hardware co-simulation block is in effect producing the clock signal for the FPGA hardware only when Simulink awakes it, the overhead associated with the rest of the Simulink model's simulation, and the communication overhead (e.g. bus latency) between Simulink and the FPGA board can significantly limit the performance achieved by the hardware. As long as the amount of computation inside the FPGA is significant with respect to the communication overhead (e.g. the amount of logic is large, or the hardware is significantly over-clocked), the hardware will provide significant simulation speed-up.

Free-Running Clock

In free-running clock mode, the hardware runs asynchronously relative to the software simulation. Unlike the single-step clock mode, where Simulink effectively generates the FPGA clock, in free-running mode, the hardware clock runs continuously inside the FPGA itself. In this mode, simulation is not bit and cycle true to the original model, because Simulink is only sampling the internal state of the hardware at the times when Simulink awakes the hardware co-simulation block. The FPGA port I/O is no longer synchronized with events in Simulink. When an event occurs on a Simulink port, the value is either read from or written to the corresponding port in hardware at that time. However, since an unknown number of clock cycles have elapsed in hardware between port events, the current state of the hardware cannot be reconciled to the original Model Composer model. For many streaming applications, this is in fact highly desirable, as it allows the FPGA to work at full speed, synchronizing only periodically to Simulink.

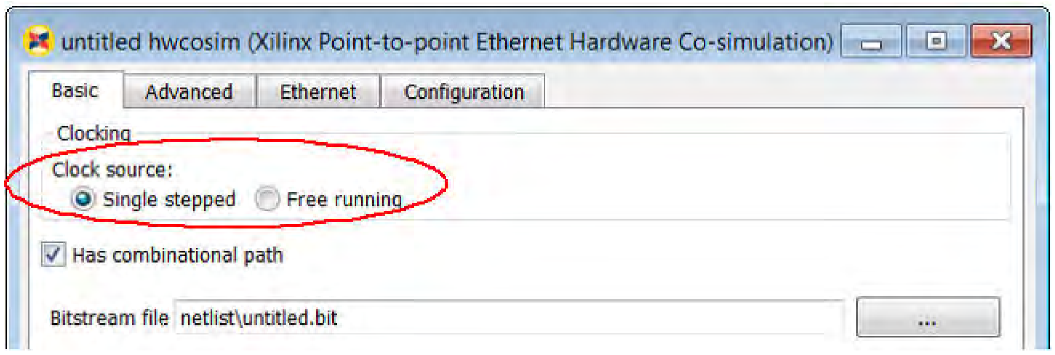
In free-running mode, you must build explicit synchronization mechanisms into the Model Composer model. A simple example is a status register, exposed as an output port on the hardware co-simulation block, which is set in hardware when a condition is met. The rest of the Model Composer model can poll the status register to determine the state of the hardware.

Selecting the Clock Mode

Not every hardware board supports a free-running clock. However, for those that do, the parameters dialog box for the hardware co-simulation block provides a means to select the desired clocking mode. You may change the co-simulation clocking mode before simulation starts by selecting either the **Single stepped** or **Free running** radio button for **Clock Source** in the parameters dialog box.

***Note*:** The clocking options available to a hardware co-simulation block depend on the FPGA board being used (i.e., some boards may not support a free-running clock source, in which case it is not available as a dialog box parameter).

Figure 123: **Single Stepped Button**


untitled hwcosim (Xilinx Point-to-point Ethernet Hardware Co-simulation)
Basic
Advanced
Ethernet
Configuration
Clocking
...

Clock source: [✓ Single stepped] Free running

[✓] Has combinational path

Bitstream file netlist\untitled.bit

For a description of a way to programmatically turn on or off a free-running clock using M-Hardware Cosim, see the description of the Run operation under in M-Hwcosim MATLAB Class.

Burst Data Transfers for Hardware Co-Simulation

Hardware co-simulation (HWCosim) is a methodology by which a user can offload, either partially or whole, the most compute intensive portion of a model into the actual target FPGA platform. The host system provides the stimulus to the model via the co-simulation interface (typically JTAG) and post-processes the response. This methodology is useful for validating the correctness of the generated hardware design on the target platform itself, as well as for speeding up the simulation time during verification of the model in a hardware co-verification scenario.

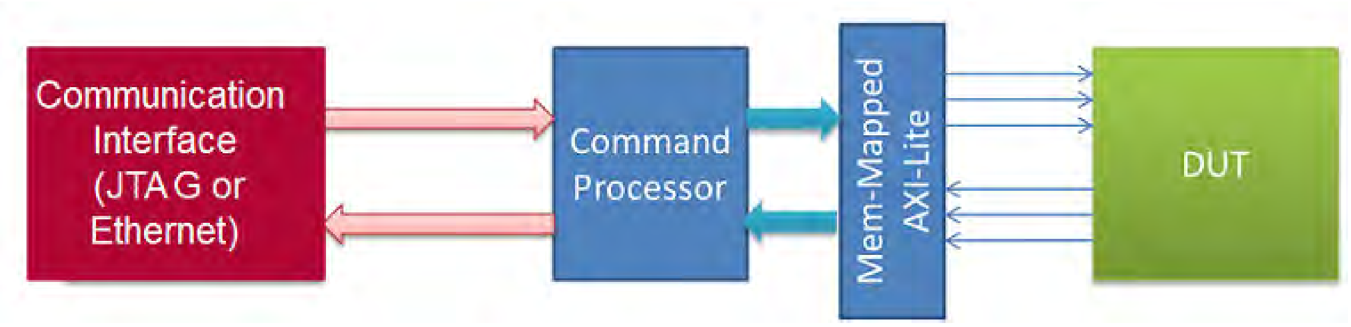
MATLAB/Simulink in conjunction with Model Composer currently supports two variants of HWCosim: GUI-based and MATLAB M-script based. The first is run under the control of the Simulink scheduler, and can only progress one clock cycle at a time, due to the potential for feedback loops in the model.

The second variant is MATLAB M-script based simulation under Model Composer control (M-HWCosim), which is commonly used in testbenches produced as collateral during the bitstream generation from the System Generator token. These testbenches are typically feedback-free and come with a-priori known input that can be transferred to the device in larger batches.

Hardware Co-Simulation Overview

A high-level overview of hardware co-simulation (HWCosim) is given in the figure below. At the center of it is the device under test (DUT). The DUT is typically a piece of IP that is developed and tested within a Simulink test framework providing the stimulus and receiving (and potentially evaluating) the response. In order to allow for Simulink to communicate with the DUT it needs to be embedded into the HWCosim wrapper consisting of the following components:

Figure 124: **Hardware Co-Simulation Flow**

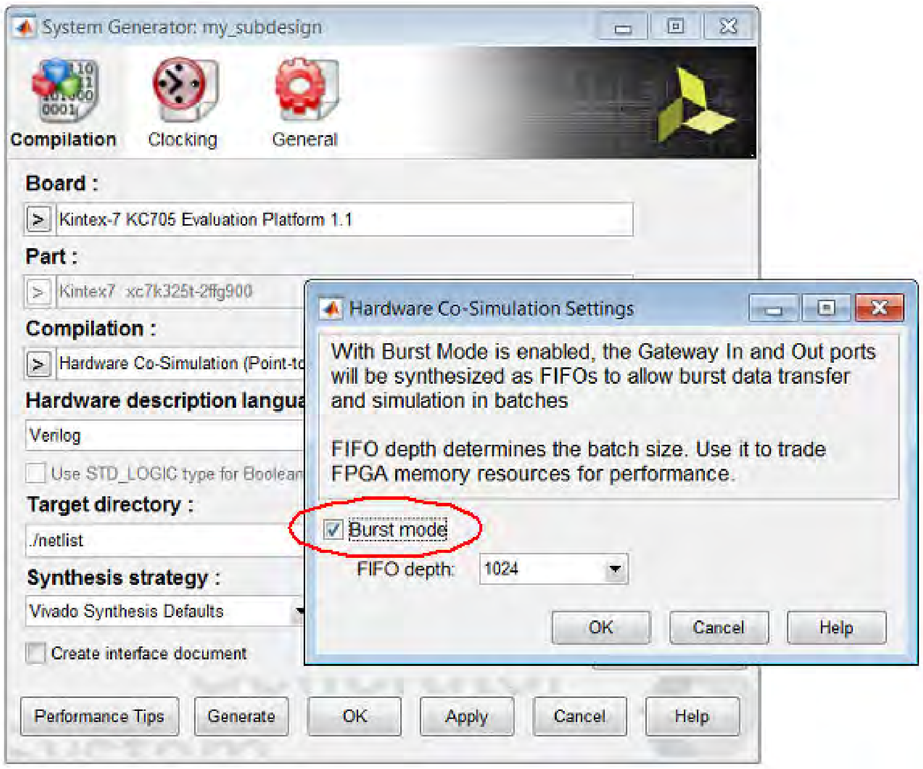


- **Communication interface (JTAG):** Used for communications with the host PC, receiving the command messages and sending responses.
- **Command processor:** Command messages are parsed and executed.
- **Memory-mapped AXI4-Lite register bank:** Use `write` commands to set up the stimulus data in the register map, which is driving the inputs to the DUT. Similarly, use `read` commands to query the memory-mapped DUT outputs. Finally, use a `run(x)` command to the memory-mapped clock control register to trigger exactly "x" clock pulses on the DUT's clock input. Alternatively, use `run(inf)` to start the free-running clock mode and `run(0)` to turn the clock off.

Burst Data Transfer Mode

If you enable burst data transfer mode in the System Generator token (**Compilation → Settings → Burst mode**), the non-clock input and output registers will be replaced with "n"-entry FIFOs. You can select "n" (**FIFO depth**), which is useful for trading off performance versus FPGA block RAM resource use.

Figure 125: **Burst Mode**



Board : Kintex-7 KC705 Evaluation Platform 1.1

Part : Kintex7 xc7k325t-2ffg900

Compilation : Hardware Co-Simulation (Point-to

Hardware description langua Verilog

☐ Use STD_LOGIC type for Boolean

Target directory : ./netlist

Synthesis strategy : Vivado Synthesis Defaults

☐ Create interface document

With Burst Mode is enabled, the Gateway In and Out ports will be synthesized as FIFOs to allow burst data transfer and simulation in batches

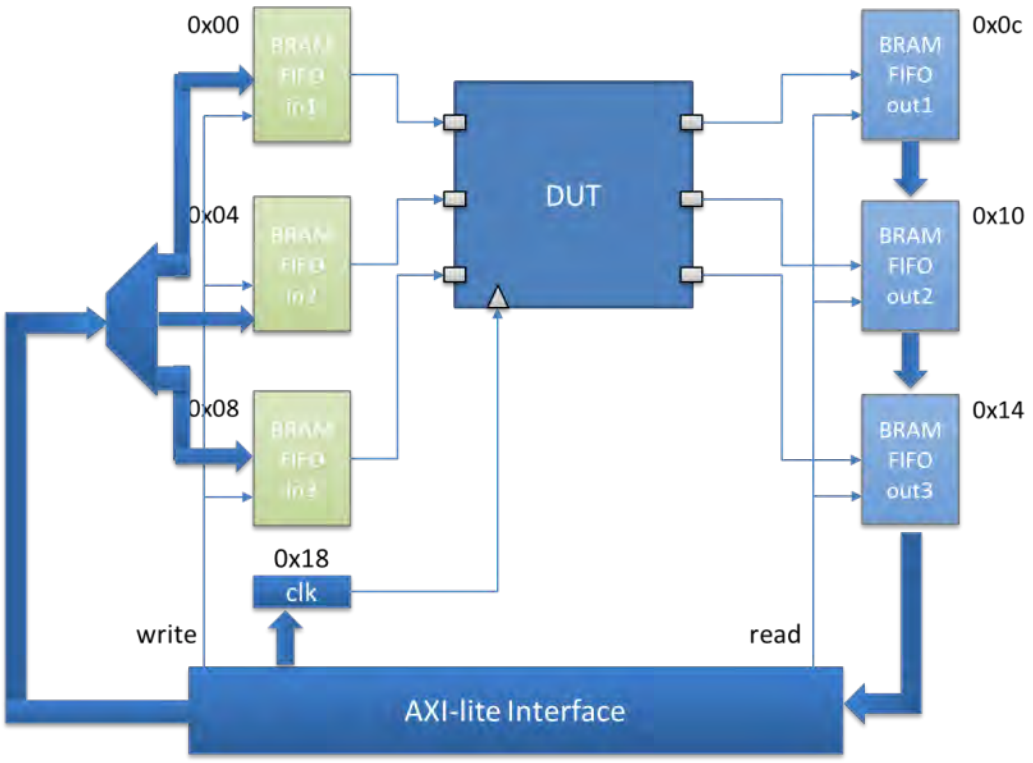
FIFO depth determines the batch size. Use it to trade FPGA memory resources for performance.

☑ Burst mode

FIFO depth: 1024

Enabling **Burst mode** allows the M-HWCosim scheduler to "burst write" a time-sequence of "n" values into each input FIFO, run the clock for a number of cycles determined by the rate of input/output ports and the FIFO depths, and capture the resulting output in the output FIFOs. After the batch has been run, the scheduler proceeds to "burst read" the contents of the output FIFOs into a MATLAB array, where it can be checked against expected data.

Figure 126: **Burst Mode Flow**



This batch processing of time samples allows to better pack data into JTAG sequences or thereby significantly reducing overhead.

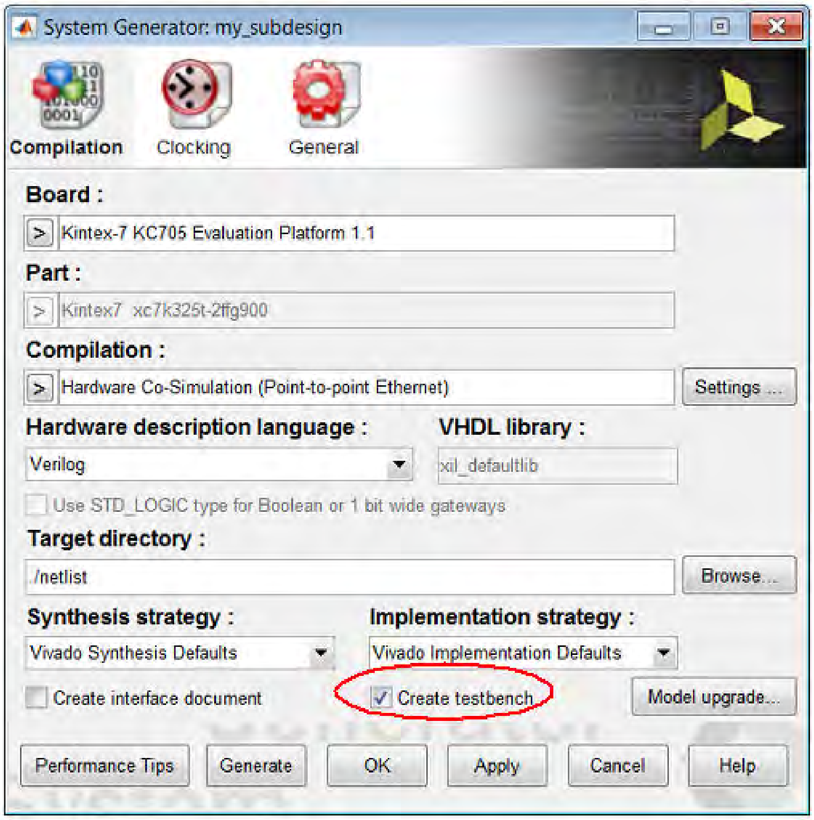
How to Use Burst Data Transfer Mode

The simplest way for you to start using burst data transfer mode is via an automatically generated test bench script. Advanced users can make use of the HWCosim API exposed via the MATLAB Hwcosim objects that are shipped with Model Composer.

Automatic Testbench Generation

Testbench generation is run alongside the hardware co-simulation compilation flow. Open the System Generator token in the Simulink model and wait for the dialog box to appear. The first tab shows the **Compilation** options. A drop-down list shows the available compilation targets. After selecting one of the two hardware co-simulation flows (depending on which one is available for the selected board), the **Settings** button will be enabled and when selected it will open a secondary dialog box where burst mode and the desired FIFO depth can be chosen. After burst mode has been turned on, you can enable the automatic creation of an M-HWCosim test bench script by enabling **Create testbench** at the bottom of the **Compilation** tab.

Figure 127: **Create Test Bench**


System Generator: my_subdesign

Board : Kintex-7 KC705 Evaluation Platform 1.1

Part : Kintex7 xc7k325t-2ffg900

Compilation : Hardware Co-Simulation (Point-to-point Ethernet)

Hardware description language : Verilog

VHDL library : xil_defaultlib

☐ Use STD_LOGIC type for Boolean or 1 bit wide gateways

Target directory : ./netlist

Synthesis strategy : Vivado Synthesis Defaults

Implementation strategy : Vivado Implementation Defaults

☐ Create interface document

✓ Create testbench

The test generator will produce this M-script file in the Target Directory:

```
<design_name>_<sub_system>_hwcosim_test.m
```

You can run this script from the MATLAB® console. The script will also run the Simulink model to determine the stimulus data driven to the Xilinx® Gateway In blocks (from the other Simulink source blocks or MATLAB variables), while also capturing the expected output produced by the Xilinx® Block Design (BD) and exporting the data to the **Target directory** as these separate data files:

```
<design_name>_<sub_system>_<port_name>.dat
```

To run the test bench, you can open the MATLAB console, change directory to the Target Directory, and run the script by name. If the test fails this will be printed on the console, and the failing comparisons will be listed in this file:

```
<design_name>_<sub_system>_hwcosim_test.result
```

Burst Mode Testbench Script

The following is a test bench generated for an example design as part of the compilation flow:

```
%% project3_burst_hwcosim_test
% project3_burst_hwcosim_test is an automatically generated example MCode
% function that can be used to open a hardware co-simulation (hwcosim)
target,
% load the bitstream, write data to the hwcosim target's input blocks, fetch
% the returned data, and verify that the test passed. The returned value of
% the test is the amount of time required to run the test in seconds.
% Fail / Pass is indicated as an error or displayed in the command window.

%%
% PLEASE NOTE that this file is automatically generated and gets re-created
% every time the Hardware Co-Simulation flow is run. If you modify any part
% of this script, please make sure you save it under a new name or in a
% different location.

%%
% The following sections exist in the example test function:
% Initialize Bursts
% Initialize Input Data & Golden Vectors
% Open and Simulate Target
% Release Target on Error
% Test Pass / Fail

function eta = project3_burst_hwcosim_test
eta = 0;

%%
% ncycles is the number of cycles to simulate for and should be adjusted if
% the generated testbench simulation vectors are substituted by user data.
ncycles = 10;

%%
% Initialize Input Data & Golden Vectors
% xlHwcosimTestbench is a utility function that reformats fixed-point HDL
Netlist
% testbench data vectors into a double-precision floating-point MATLAB
binary
% data array.
xlHwcosimTestbench('.','project3_burst');

%%
% The testbench data vectors are both stimulus data for each input port, as
% well as expected (golden) data for each output port, recorded during the
% Simulink simulation portion of the Hardware Co-Simulation flow.
% Data gets loaded from the data file ('<name>_<port>_hwcosim_test.dat')
% into the corresponding 'testdata_<port>' workspace variables using
% 'getfield(load('<name>_<port>_hwcosim_test.dat' ... ' commands.
%
% Alternatively, the workspace variables holding the stimulus and / or
golden
% data can be assigned other data (including dynamically generated data) to
% test the design with. If using alternative data assignment, please make
% sure to adjust the "ncycles" variable to the proper number of cycles, as
% well as to disable the "Test Pass / Fail" section if unused.
testdata_noise_x0 =
getfield(load('project3_burst_noise_x0_hwcosim_test.dat', '-mat'),
'values');
testdata_scale = getfield(load('project3_burst_scale_hwcosim_test.dat', '-
mat'), 'values');
testdata_wave = getfield(load('project3_burst_wave_hwcosim_test.dat', '-
mat'), 'values');
testdata_intout = getfield(load('project3_burst_intout_hwcosim_test.dat', '-
mat'), 'values');
testdata_sigout = getfield(load('project3_burst_sigout_hwcosim_test.dat', '-
mat'), 'values');

%%
% The 'result_<port>' workspace variables are arrays to receive the actual
results
% of a Hardware Co-Simulation read from the FPGA. They will be compared to
the
% expected (golden) data at the end of the Co-Simulation.
result_intout = zeros(size(testdata_intout));
result_sigout = zeros(size(testdata_sigout));

%%
% project3_burst.hwc is the data structure containing the Hardware Co-
Simulation
% design information returned after netlisting the Simulink / System
% Generator model.
% Hwcosim(project) instantiates and returns a handle to the API shared
library object.
project = 'project3_burst.hwc';
h = Hwcosim(project);
try
    %% Open the Hardware Co-Simulation target and co-simulate the design
    open(h);
    cosim_t_start = tic;
    h('noise_x0') = testdata_noise_x0;
    h('scale') = testdata_scale;
    h('wave') = testdata_wave;
    run(h, ncycles);
    result_intout = h('intout');
    result_sigout = h('sigout');
    eta = toc(cosim_t_start);
    % Release the handle for the Hardware Co-Simulation target
    release(h);

%% Release Target on Error
catch err
    release(h);
    rethrow(err);
    error('Error running hardware co-simulation testbench. Please refer to
hwcosim.log for
details.');
end

%% Test Pass / Fail
logfile = 'project3_burst_hwcosim_test.results';
logfd = fopen(logfile, 'w');
sim_ok = true;
sim_ok = sim_ok & xlHwcosimCheckResult(logfd, 'intout', testdata_intout,
result_intout);
sim_ok = sim_ok & xlHwcosimCheckResult(logfd, 'sigout', testdata_sigout,
result_sigout);
fclose(logfd);
if ~sim_ok
    error('Found errors in the simulation results. Please refer to
project3_burst_hwcosim_test.results for details.');
end
disp(['Hardware Co-Simulation successful. Data matches the Simulink
simulation and completed in
' num2str(eta) ' seconds.']) ;
```

This script first defines the number of cycles (`ncycles`) to run in the simulation, prepares the test bench, and loads the stimulus data and expected output into MATLAB arrays. Then it creates an Hwcosim object instance with a handle (`h`), which loads the HWCosim API shared library. Inside the try-catch block it opens the instance, initializes the FPGA, and opens a connection to it.

Once the setup phase is complete, the code between the `tic` and `toc` timing commands executes the write-run-read commands. Please note that unlike in previous versions of HWCosim, this test bench does not require a for-loop to cycle through every clock cycle. This is due to the new smart cache layer which can buffer up nearly arbitrary size write commands in host memory before issuing smaller cycles of write-run-read batches to the hardware (during execution of the user-visible `run(h, ncycles)` command).

At the end of the execution phase the HWCosim instance is released and the test bench compares actual to expected outputs.

Comments in the test bench code will help you understand the flow of the hardware co-simulation and help you develop customized test bench scripts for your design.

Importing HDL Modules

Sometimes it is important to add one or more existing HDL modules to a Model Composer design. The HDL Black Box block allows VHDL and Verilog to be brought into a design. The Black Box block behaves like other Model Composer HDL blocks - it is wired into the design, participates in simulations, and is compiled into hardware. When Model Composer compiles a Black Box block, it automatically connects the ports of the Black Box to the rest of the design. A Black Box can be configured to support either synchronous clock designs or multiple hardware clock designs based on the context and System Generator token settings.

The Black Box Interface	
Black Box HDL Requirements and Restrictions	Details the requirements and restrictions for VHDL, Verilog, and EDIF associated with black boxes.
Black Box Configuration Wizard	Describes how to use the Black Box Configuration Wizard.
Black Box Configuration M-Function	Describes how to create a black box configuration M-function.

HDL Co-Simulation	
Configuring the HDL Simulator	Explains how to configure the Vivado® simulator or Questa to co-simulate the HDL in the Black Box block.

Co-Simulating Multiple Black Boxes	Describes how to co-simulate several Black Box blocks in a single HDL simulator session.

Black Box HDL Requirements and Restrictions

An HDL component associated with a black box must adhere to the following Model Composer requirements and restrictions:

- The entity name must not collide with any other entity name in the design.
- Bi-directional ports are supported in HDL black boxes, however they will not be displayed in the Model Composer as ports; they only appear in the generated HDL after netlisting.
- For Verilog black boxes, the module and port names must follow standard HDL naming conventions.
- Any port that is a clock or clock enable must be of type std_logic. (For Verilog black boxes, ports must be of non-vector inputs, e.g., input clk.)
- Clock and clock enable ports in black box HDL should be expressed as follows: Clock and clock enables must appear as pairs (i.e., for every clock, there is a corresponding clock enable, and vice-versa). A black box may have more than one clock port and its behavior changes based on the context of the design.
    - In Synchronous single clock design context, a single clock source is used to drive each clock port. Only the clock enable rates differ.
    - In case of multiple independent hardware clock design context, two different clock sources is used to drive clock and clock enable pins.
- Each clock name (respectively, clock enable name) must contain the substring `clk`, for example `my_clk_1` and `my_ce_1`.
- The name of a clock enable must be the same as that for the corresponding clock, but with `ce` substituted for `clk`. For example, if the clock is named `src_clk_1`, then the clock enable must be named `src_ce_1`.
- Falling-edge triggered output data cannot be used.

IMPORTANT! *It is not recommended to use the black box block to import encrypted RTLs which are generated for Vivado IP. As an alternative, try to import Vivado IPs using a DCP file.*

Black Box Configuration M-Function

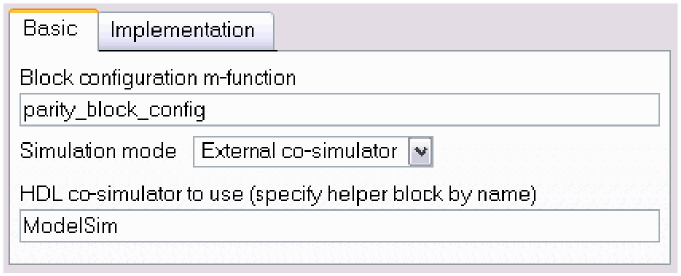
An imported module is represented in Model Composer by a Black Box block. Information about the imported module is conveyed to the black box by a configuration M-function. This function defines the interface, implementation, and the simulation behavior of the black box block it is associated with. The information a configuration M-function defines includes the following:

- Name of the top-level entity for the module

- VHDL or Verilog language selection
- Port descriptions
- Generics required by the module
- Synchronous single clock or asynchronous multiple independent clock configuration
- Clocking and sample rates
- Files associated with the module
- Whether the module has any combinational paths

The name of the configuration M-function associated with a black box is specified as a parameter in the dialog box (`parity_block_config.m`).

Figure 128: **Parameter Dialog**



Block configuration m-function: parity_block_config
Simulation mode: External co-simulator
HDL co-simulator to use (specify helper block by name): ModelSim

Configuration M-functions use an object-based interface to specify black box information. This interface defines two objects, SysgenBlockDescriptor and SysgenPortDescriptor. When Model Composer invokes a configuration M-function, it passes the function a block descriptor:

```
function sample_block_config(this_block)
```

A SysgenBlockDescriptor object provides methods for specifying information about the black box. Ports on a block descriptor are defined separately using port descriptors.

Language Selection

The black box can import VHDL and Verilog modules. SysgenBlockDescriptor provides a method, setTopLevelLanguage, that tells the black box what type of module you are importing. This method should be invoked once in the configuration M-function. The following code shows how to select between the VHDL and Verilog languages.

VHDL Module:

```
this_block.setTopLevelLanguage('VHDL');
```

Verilog Module:

```
this_block.setTopLevelLanguage('Verilog');
```

Note: The Configuration Wizard automatically selects the appropriate language when it generates a configuration M-function.

Specifying the Top-Level Entity

You must tell the black box the name of the top-level entity that is associated with it. SysgenBlockDescriptor provides a method, setEntityName, which allows you to specify the name of the top-level entity.

Note: Use lower case text to specify the entity name.

For example, the following code specifies a top-level entity named `foo`.

```
this_block.setEntityName('foo');
```

Note: The Configuration Wizard automatically sets the name of the top-level entity when it generates a configuration M-function.

Defining Port Blocks

The port interface of a black box is defined by the block's configuration M-function. Recall that black box ports are defined using port descriptors. A port descriptor provides methods for configuring various port attributes, including port width, data type, binary point, and sample rate.

Adding New Ports

When defining a black box port interface, it is necessary to add input and output ports to the block descriptor. These ports correspond to the ports on the module you are importing. In your model, the black box block port interface is determined by the port names that are declared on the block descriptor object. SysgenBlockDescriptor provides methods for adding input and output ports:

Adding an input port:

```
this_block.addSimulinkInport('din');
```

Adding an output port:

```
this_block.addSimulinkOutport('dout');
```

The string parameter passed to methods addSimulinkInport and addSimulinkOutport specifies the port name. These names should match the corresponding port names in the imported module.

Note: Use lower case text to specify port names.

Adding a bidirectional port:

```
config_phase = this_block.getConfigPhaseString;
if (strcmpi(config_phase,'config_netlist_interface'))
 this_block.addInoutport('bidi');
 % Rate and type info should be added here as well
end
```

Bidirectional ports are supported only during the netlisting of a design and will not appear on the Model Composer diagram; they only appear in the generated HDL. As such, it is important to only add the bi-directional ports when Model Composer is generating the HDL. The if-end conditional statement is guarding the execution of the code to add-in the bi-directional port.

It is also possible to define both the input and output ports using a single method call. The setSimulinkPorts method accepts two parameters. The first parameter is a cell array of strings that define the input port names for the block. The second parameter is a cell array of strings that define the output port names for the block.

Note: The Configuration Wizard automatically sets the port names when it generates a configuration M-function.

Obtaining a Port Object

Once a port has been added to a block descriptor, it is often necessary to configure individual attributes on the port. Before configuring the port, you must obtain a descriptor for the port you would like to configure. SysgenBlockDescriptor provides methods for accessing the port objects that are associated with it. For example, the following method retrieves the port named `din` on the `this_block` descriptor:

Accessing a SysgenPortDescriptor object:

```
din = this_block.port('din');
```

In the above code, an object `din` is created and assigned to the descriptor returned by the `port` function call.

SysgenBlockDescriptor also provides methods, `inport` and `outport`, that return a port object given a port index. A port index is the index of the port (in the order shown on the block interface) and is some value between 1 and the number of input/output ports on the block. These methods are useful when you need to iterate through the block's ports (e.g., for error checking).

Configuring Port Types

SysgenPortDescriptor provides methods for configuring individual ports. For example, assume port dout is unsigned, 12 bits, with binary point at position 8. The code below shows one way in which this type can be defined.

```
dout = this_block.port('dout');
dout.setWidth(12);
dout.setBinPt(8);
dout.makeUnsigned();
```

The following also works:

```
dout = this_block.port('dout');
dout.setType('Ufix_12_8');
```

The first code segment sets the port attributes using individual method calls. The second code segment defines the signal type by specifying the signal type as a string. Both code segments are functionally equivalent.

The black box supports HDL modules with 1-bit ports that are declared using either single bit port (e.g., std_logic) or vectors (e.g., std_logic_vector(0 downto 0)) notation. By default, Model Composer assumes ports to be declared as vectors. You may change the default behavior using the useHDLVector method of the descriptor. Setting this method to `true` tells Model Composer to interpret the port as a vector. A `false` value tells Model Composer to interpret the port as single bit.

```
dout.useHDLVector(true); % std_logic_vector
dout.useHDLVector(false); % std_logic
```

***Note*:** The Configuration Wizard automatically sets the port types when it generates a configuration M-function.

Configuring Bi-Directional Ports for Simulation

Bidirectional ports (or inout ports) are supported only during the generation of the HDL netlist, that is, bi-directional ports will not show up in the Model Composer diagram. By default, bi-directional ports will be driven with 'X' during simulation. It is possible to overwrite this behavior by associating a data file to the port. Be sure to guard this code because bi-directional ports can only be added to a block during the `config_netlist_interface` phase.

```
if (strcmpi(this_block.getConfigPhaseString,'config_netlist_interface'))
  bidi_port = this_block.port('bidi');
  bidi_port.setGatewayFileName('bidi.dat');
end
```

In the above example, a text file, `bidi.dat`, is used during simulation to provide stimulation to the port. The data file should be a text file, where each line represents the signal driven on the port at each simulation cycle. For example, a 3-bit bi-directional port that is simulated for 4 cycles might have the following data file:

```
ZZZ
110
011
XXX
```

Simulation will return with an error if the specified data file cannot be found.

Configuring Port Sample Rates

The Black Box block supports ports that have different sample rates. By default, the sample rate of an output port is the sample rate inherited from the input port (or ports, if the inputs run at the same sample rate). Sometimes, it is necessary to explicitly specify the sample rate of a port (e.g., if the output port rate is different than the block's input sample rate).

***Note*:** When the inputs to a black box have different sample rates, you must specify the sample rates of every output port.

SysgenPortDescriptor provides a method called setRate that allows you to explicitly set the rate of a port.

***Note*:** The rate parameter passed to the setRate method is not necessarily the Simulink® sample rate that the port runs at. Instead, it is a positive integer value that defines the ratio between the desired port sample period and the Simulink® system clock period defined by the System Generator token dialog box.

Assume you have a model in which the Simulink system period value for the model is defined as 2 sec. Also assume that the example `dout` port is assigned a rate of 3 by invoking the setRate method as follows:

```
dout.setRate(3);
```

A rate of 3 means that a new sample is generated on the dout port every 3 Simulink system periods. Because the Simulink system period is 2 sec, this means the Simulink sample rate of the port is 3 x 2 = 6 sec.

***Note*:** If your port is a non-sampled constant, you can define it in the configuration M-function using the setConstant method of SysgenPortDescriptor. You can also define a constant by passing Inf to the setRate method.

Dynamic Output Ports

A useful feature of the black box is its ability to support dynamic output port types and rates. For example, it is often necessary to set an output port width based on the width of an input port. SysgenPortDescriptor provides member variables that allow you to determine the configuration of a port. You can set the type or rate of an output port by examining these member variables on the block's input ports.

For example, you can obtain the width and rate of a port (in this case `din`) as follows:

```
input_width = this_block.port('din').width;
input_rate  = this_block.port('din').rate;
```

Note: A black box's configuration M-function is invoked at several different times when a model is compiled. The configuration function may be invoked before the data types and rates have been propagated to the black box.

The SysgenBlockDescriptor object provides Boolean member variables `inputTypesKnown` and `inputRatesKnown` that tell whether the port types and rates have been propagated to the block. If you are setting dynamic output port types or rates based on input port configurations, the configuration calls should be nested inside conditional statements that check that values of `inputTypesKnown` and `inputRatesKnown`.

The following code shows how to set the width of a dynamic output port `dout` to have the same width as input port `din`:

```
if (this_block.inputTypesKnown)
  dout.setWidth(this_block.port('din').width);
end
```

Setting dynamic rates works in a similar manner. The code below sets the sample rate of output port `dout` to be twice as slow as the sample rate of input port `din`:

```
if (this_block.inputRatesKnown)
  dout.setRate(this_block.port('din').rate*2);
end
```

Black Box Clocking

In order to import a multirate module, you must tell Model Composer information about the module's clocking in the configuration M-function. Model Composer treats clock and clock enables differently than other types of ports. A clock port on an imported module must always be accompanied by a clock enable port (and vice versa). In other words, clock and clock enables must be defined as a pair, and exist as a pair in the imported module. This is true for both single rate and multirate designs.

Although clock and clock enables must exist as pairs, Model Composer drives all clock ports on your imported module with the FPGA system clock. The clock enable ports are driven by clock enable signals derived from the FPGA system clock.

SysgenBlockDescriptor provides a method, `addClkCEPair`, which allows you to define clock and clock enable information for a black box. This method accepts three parameters. The first parameter defines the name of the clock port (as it appears in the module). The second parameter defines the name of the clock enable port (also as it appears in the module).

The port names of a clock and clock enable pair must follow the naming conventions provided below:

- The clock port must contain the substring `clk`
- The clock enable must contain the substring `ce`
- The strings containing the substrings `clk` and `ce` must be the same (e.g. `my_clk_1` and `my_ce_1`).

The third parameter defines the rate relationship between the clock and the clock enable port. The rate parameter should not be thought of as a Simulink sample rate. Instead, this parameter tells Model Composer the relationship between the clock sample period, and the desired clock enable sample period. The rate parameter is an integer value that defines the ratio between the clock rate and the corresponding clock enable rate.

For example, assume you have a clock enable port named `ce_3` that would like to have a period three times larger than the system clock period. The following function call establishes this clock enable port:

```
addClkCEPair('clk_3','ce_3',3);
```

When Model Composer compiles a black box into hardware, it produces the appropriate clock enable signals for your module, and automatically wires them up to the appropriate clock enable ports.

Combinational Paths

If the module you are importing has at least one combinational path (i.e. a change on any input can effect an output port without a clock event), you must indicate this in the configuration M-function. SysgenBlockDescriptor object provides a `tagAsCombinational` method that indicates your module has a combinational path. It should be invoked as follows in the configuration M-function:

```
this_block.tagAsCombinational;
```

Specifying VHDL Generics and Verilog Parameters

You may specify a list of generics that get passed to the module when Model Composer compiles the model into HDL. Values assigned to these generics can be extracted from mask parameters and from propagated port information (e.g. port width, type, and rate). This flexible means of generic assignment allows you to support highly parametric modules that are customized based on the Simulink environment surrounding the black box.

The `addGeneric` method allows you to define the generics that should be passed to your module when the design is compiled into hardware. The following code shows how to set a VHDL Integer generic, `dout_width`, to a value of 12.

```
addGeneric('dout_width','Integer','12');
```

It is also possible to set generic values based on port on propagated input port information (e.g. a generic specifying the width of a dynamic output port).

Because a black box's configuration M-function is invoked at several different times when a model is compiled, the configuration function may be invoked before the data types (or rates) have been propagated to the black box. If you are setting generic values based on input port types or rates, the `addGeneric` calls should be nested inside a conditional statement that checks the value of the `inputTypesKnown` or `inputRatesKnown` variables. For example, the width of the dout port can be set based on the value of din as follows:

```
if (this_block.inputTypesKnown)
  % set generics that depend on input port types
  this_block.addGeneric('dout_width', ...
  this_block.port('din').width);
end
```

Generic values can be configured based on mask parameters associated with a block box. SysgenBlockDescriptor provides a member variable, `blockName`, which is a string representation of the black box's name in Simulink. You may use this variable to gain access the black box associated with the particular configuration M-function. For example, assume a black box defines a parameter named `init_value`. A generic with name `init_value` can be set as follows:

```
simulink_block = this_block.blockName;
init_value = get_param(simulink_block,'init_value');
this_block.addGeneric('init_value', 'String', init_value);
```

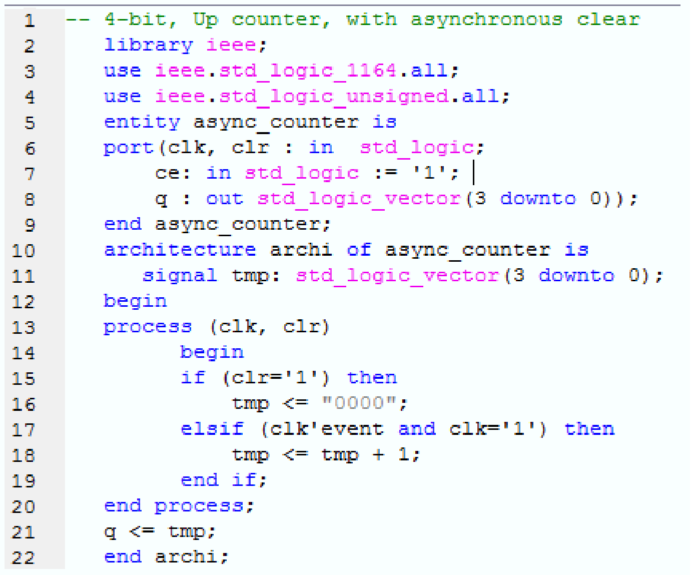
Note: You can add your own parameters (e.g. values that specify generic values) to the black box by doing the following:

- Copy a black box into a Simulink library or model.
- Break the link on the black box.
- Add the desired parameters to the black box dialog box.

Black Box VHDL Library Support

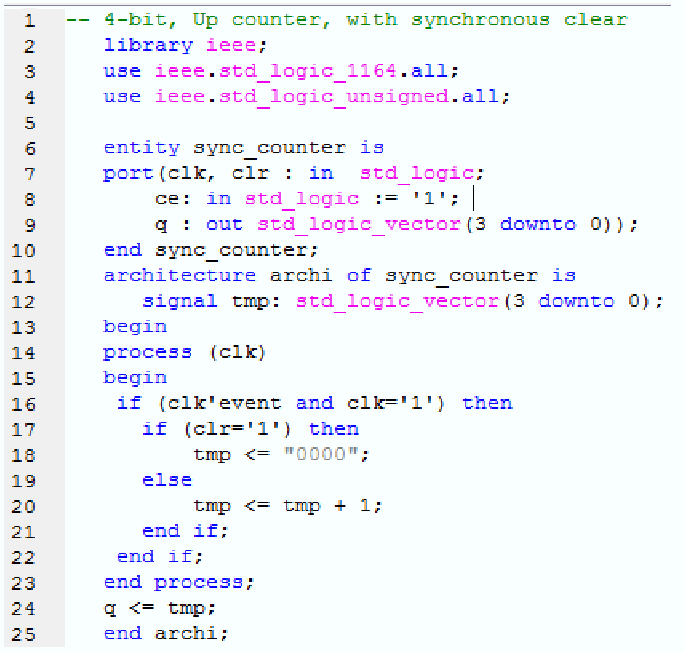
This Black Box feature allows you to import VHDL modules that have predefined library dependencies. The following example illustrates how to do this import.

The VHDL module below is a 4-bit, Up counter with asynchronous clear (`async_counter.vhd`). It will be compiled into a library named `async_counter_lib`.



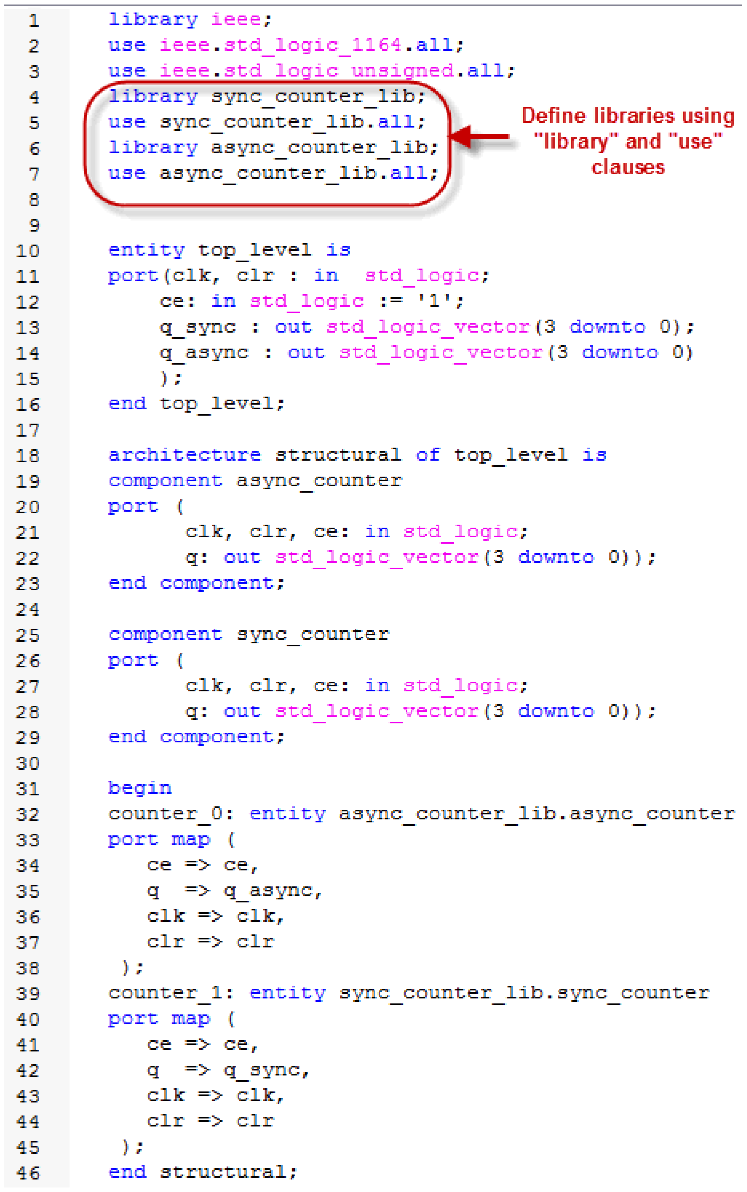
```
-- 4-bit, Up counter, with asynchronous clear
   library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_unsigned.all;
   entity async_counter is
   port(clk, clr : in  std_logic;
       ce: in std_logic := '1';
       q : out std_logic_vector(3 downto 0));
   end async_counter;
   architecture archi of async_counter is
      signal tmp: std_logic_vector(3 downto 0);
   begin
   process (clk, clr)
         begin
         if (clr='1') then
             tmp <= "0000";
         elsif (clk'event and clk='1') then
             tmp <= tmp + 1;
         end if;
   end process;
   q <= tmp;
   end archi;
```

The VHDL module below is a 4-bit, Up counter with synchronous clear (`sync_counter.vhd`). It will be compiled into a library named `sync_counter_lib`.



```
-- 4-bit, Up counter, with synchronous clear
   library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_unsigned.all;

   entity sync_counter is
   port(clk, clr : in  std_logic;
       ce: in std_logic := '1';
       q : out std_logic_vector(3 downto 0));
   end sync_counter;
   architecture archi of sync_counter is
      signal tmp: std_logic_vector(3 downto 0);
   begin
   process (clk)
   begin
    if (clk'event and clk='1') then
      if (clr='1') then
          tmp <= "0000";
      else
          tmp <= tmp + 1;
      end if;
    end if;
   end process;
   q <= tmp;
   end archi;
```

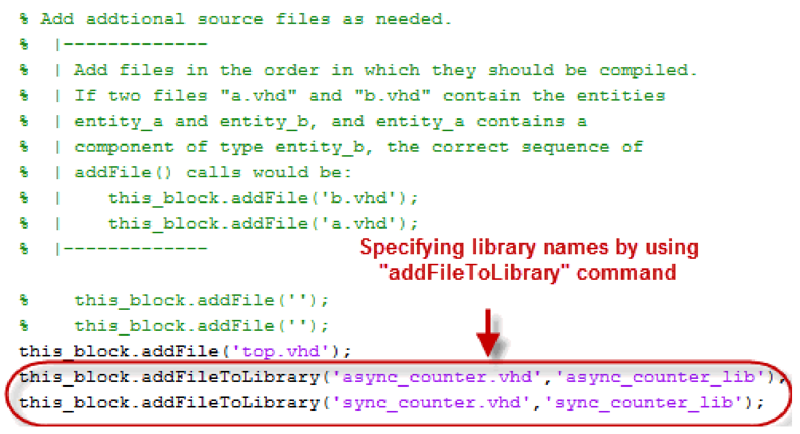
The VHDL module below is the top-level module that is used to instantiate the previous modules. This is the module that you need to point to when adding the BlackBox into your Model Composer model.



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
library sync_counter_lib;
use sync_counter_lib.all;
library async_counter_lib;
use async_counter_lib.all;


entity top_level is
port(clk, clr : in  std_logic;
    ce: in std_logic := '1';
    q_sync : out std_logic_vector(3 downto 0);
    q_async : out std_logic_vector(3 downto 0)
    );
end top_level;

architecture structural of top_level is
component async_counter
port (
      clk, clr, ce: in std_logic;
      q: out std_logic_vector(3 downto 0));
end component;

component sync_counter
port (
      clk, clr, ce: in std_logic;
      q: out std_logic_vector(3 downto 0));
end component;

begin
counter_0: entity async_counter_lib.async_counter
port map (
   ce => ce,
   q  => q_async,
   clk => clk,
   clr => clr
 );
counter_1: entity sync_counter_lib.sync_counter
port map (
   ce => ce,
   q  => q_sync,
   clk => clk,
   clr => clr
 );
end structural;
```

The VHDL is imported by first importing the top-level entity, `top_level`, using the Black Box.

Once the file is imported, the associated Black Box Configuration M-file needs to be modified as follows:



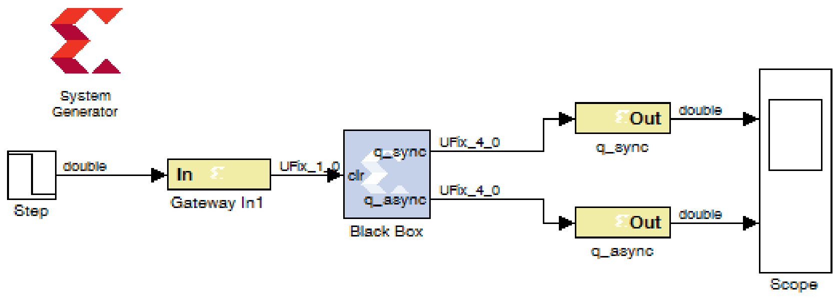
```
% Add addtional source files as needed.
%  |-------------
%  | Add files in the order in which they should be compiled.
%  | If two files "a.vhd" and "b.vhd" contain the entities
%  | entity_a and entity_b, and entity_a contains a
%  | component of type entity_b, the correct sequence of
%  | addFile() calls would be:
%  |    this_block.addFile('b.vhd');
%  |    this_block.addFile('a.vhd');
%  |-------------
%    this_block.addFile('');
%    this_block.addFile('');
this_block.addFile('top.vhd');
this_block.addFileToLibrary('async_counter.vhd','async_counter_lib');
this_block.addFileToLibrary('sync_counter.vhd','sync_counter_lib');
```

Specifying library names by using "addFileToLibrary" command

The interface function `addFileToLibrary` is used to specify a library name other than "work" and to instruct the tool to compile the associated HDL source to the specified library.

The Model Composer model should look similar to the following figure.

Figure 129: **Model Composer Model with Black Box Support**



The next step is to double-click the System Generator token and click the **Generate** button to generate the HDL netlist.

During the generation process, a Vivado IDE project (`.xpr`) is created and placed with the `hdl_netlist` folder under the `netlist` folder. If you double-click the Vivado IDE project and select the Libraries tab under the Source view, you will see not only a `work` library, but an `async_counter_lib` library and `sync_counter_lib` library as well.

Error Checking

It is often necessary to perform error checking on the port types, rates, and mask parameters of a black box. SysgenBlockDescriptor provides a method, setError, that allows you to specify an error message that is reported to the user. The error message that a user sees is the string parameter passed to setError.

Black Box API

SysgenBlockDescriptor Member Variables

Type	Member	Description
String	entityName	Name of the entity or module.
String	blockName	Name of the black box block.
Integer	numSimulinkInports	Number of input ports on black box.
Integer	numSimulinkOutports	Number of output ports on the black box.
Boolean	inputTypesKnown	true if all input types are defined, and false otherwise.
Boolean	inputRatesKnown	true if all input rates are defined, and false otherwise.
Array of Doubles	inputRates	Array of sample periods for the input ports (indexed as in inport(indx)). Sample period values are expressed as integer multiples of the Simulink System Period value specified by the master System Generator token
Boolean	error	true if an error has been detected, and false otherwise.
Cell Array of Strings	errorMessages	Array of all error messages for this block.

SysgenBlockDescriptor Methods

Method	Description
setTopLevelLanguage(language)	Declares language for the top-level entity (or module) of the black box. The language should be `VHDL` or `Verilog`.
setEntityName(name)	Sets name of the entity or module.
addSimulinkInport(pname)	Adds an input port to the black box. pname defines the name the port should have.
addSimulinkOutport(pname)	Adds an output port to the black box. pname defines the name the port should have.
setSimulinkPorts(in,out)	Adds input and output ports to the black box. in (respectively, out) is a cell array whose element tell the names to use for the input (resp., output) ports.
addInoutport(pname)	Adds a bidirectional port to the black box. pname defines the name the port should have. Bidirectional ports can only be added during the config_netlist_interface phase of configuration.
tagAsCombinational()	Indicate that the block has a combinational path (i.e., direct feedthrough) from an input port to an output port.
addClkCEPair(clkPname, cePname, rate)	Defines a clock/clock enable port pair for the block. clkPname and cePname tell the names for the clock and clock enable ports respectively. rate, a double, tells the rate at which the port pair runs. The rate must be a positive integer. Note the clock (respectively, clock enable) name must contain the substring clk (resp., ce). The names must be parallel in the sense that the clock enable name is obtained from the clock name by replacing clk with ce.

Method	Description
port(name)	Returns the SysgenPortDescriptor that matches the specified name.
inport(indx)	Returns the SysgenPortDescriptor that describes a given input port. indx tells the index of the port to look for, and should be between 1 and numInputPorts.
outport(indx)	Returns the SysgenPortDescriptor that describes a given output port. indx tells the index of the port to look for, and should be between 1 and numOutputPorts.
addGeneric(identifier, value)	Defines a generic (or parameter if using Verilog) for the block. identifier is a string that tells the name of the generic. value can be a double or a string. The type of the generic is inferred from value's type. If value is an integral double (e.g., 4.0), the type of the generic is set to integer. For a non-integral double, the type is set to real. When value is a string containing only zeros and ones (e.g., `0101'), the type is set to bit_vector. For any other string value, the type is set to string.
addGeneric(identifier, type, value)	Explicitly specifies the name, type, and value for a generic (or parameter, if using Verilog) for the block. All three arguments are strings. identifier tells the name, type tells the type, and value tells the value.
addFile(fn)	Adds a file name to the list of files associated to this black box, `fn` is the file name. Ordinarily, HDL files are associated to black boxes, but any sorts of files are acceptable. VHDL file names should end in `.vhd`; Verilog file names should end in `.v`. The order in which file names are added is preserved, and becomes the order in which HDL files are compiled. File names can be absolute or relative. Relative file names are interpreted with respect to the location of the .mdl or library .mdl for the design.
getDeviceFamilyName()	Gets the name of the FPGA corresponding to the black box.
getConfigPhaseString	Returns the current configuration phase as a string. A valid return string includes: config_interface, config_rate_and_type, config_post_rate_and_type, config_simulation, config_netlist_interface, and config_netlist.
setSimulatorCompilationScript(script)	Overrides the default HDL co-simulation compilation script that the black box generates. `script` tells the name of the script to use. For example, this method can be used to short-circuit the compilation phase for repeated simulations where the HDL for the black box remains unchanged.
setError(message)	Indicates that an error has occurred, and records the error message. `message` gives the error message.

SysgenPortDescriptor Member Variables

Type	Member	Description
String	name	Tells the name of the port.
Integer	simulinkPortNumber	Tells the index of this port in Simulink®. Indexing starts with 1 (as in Simulink).
Boolean	typeKnown	True if this port's type is known, and false otherwise.
String	type	Type of the port, such as UFix_<n>_<b>, Fix_<n>_<b>, or Bool.

Type	Member	Description
Boolean	isBool	True if port type is Bool, and false otherwise.
Boolean	isSigned	True if type is signed, and false otherwise.
Boolean	isConstant	True if port is constant, and false otherwise.
Integer	width	Tells the port width.
Integer	binpt	Tells the binary point position, which must be an integer in the range 0..width.
Boolean	rateKnown	True if the rate is known, and false otherwise.
Double	rate	Tells the port sample time. Rates are positive integers expressed as MATLAB® doubles. A rate can also be infinity, indicating that the port outputs a constant.

SysgenPortDescriptor Methods

Method	Description
setName(name)	Sets the HDL name to be used for this port.
setSimulinkPortNumber(num)	Sets the index associated with this port in Simulink®. num tells the index to assign. Indexing starts with 1 (as in Simulink).

setType(typeName)	Sets the type of this port to type. Type must be one of Bool, UFix_<n>_<b> , Fix_<n>_<b> , signed or unsigned. The last two choices leave the width and binary point position unchanged.
	XFloat_<exponent_bit_width>_fraction_bit_width> is also supported. For example: ap_return_port = this_block.port('ap_return');
	ap_return_port.setType('XFloat_30_2');

setWidth(w)	Sets the width of this port to w.
setBinpt(bp)	Sets the binary point position of this port to bp.
makeBool()	Makes this port Boolean.
makeSigned()	Makes this port signed.
makeUnsigned()	Makes this port unsigned.
setConstant()	Makes this port constant
setGatewayFileName(filename)	Sets the dat file name that will be used in simulations and test-bench generation for this port. This function is only meant for use with bi-directional ports so that a hand written data file can be used during simulation. Setting this parameter for input or output ports is invalid and will be ignored.
setRate(rate)	Assigns the rate for this port. rate must be a positive integer expressed as a MATLAB® double or Inf for constants.
useHDLVector(s)	Tells whether a 1-bit port is represented as single-bit (ex: std_logic) or vector (ex: std_logic_vector(0 downto 0)).
HDLTypeIsVector()	Sets representation of the 1-bit port to std_logic_vector(0 downto 0).

Multiple Independent Clock Support on Black Box

Design Rule Checks on Port Connection

When a black box is used in a multiple independent hardware clock design context, design rule checks (DRCs) for its port connections must be added in the configuration M-function. This helps to avoid invalid or incorrect port connection with different clock sources. You need to ensure all port signals are connected from/to a proper clocked-subsystem interface.

The utility `checkPortsOfSameClockDomain()` should be used to specify a list of ports from a particular clock domain and to group it together. The input arguments to this application programming interface (API) are 'SysgenBlockDescriptor' objects followed by the list of port names associated with a particular clock domain.

In the example shown below, the API puts out an error check, and verifies that the four ports are connected to the same subsystem clock domain.

```
checkPortsOfSameClockDomain (<block_descriptor>, '<port_name_1>',
'<port_name_2>',
'<port_name_3>', '<port_name_4>');
```

Configuring Port Sample Rates

In multiple clock hardware designs, the clock period of the port interface should be computed using the connected "clocked subsystem domain". By default, "synchronous system clock" source is used by all the ports, but for asynchronous clock hardware designs, it is necessary to explicitly specify the clock sources of every port (e.g., if the output port clock is different than the block's input port clock).

Note: You must set the sample rate to '1.0' for all output ports of multiple independent clock black box designs; it automatically sets the output ports to the destination clock subsystem period.

SysgenPortDescriptor provides a method called `setRate` that you can use to explicitly set the rate of a port.

Example:

```
port('<port_name>').setRate(1.0)
```

Black Box Clocking

In order to import a synchronous or asynchronous black box module, you must tell Model Composer information about the module's clocking in the configuration M-function. Model Composer treats clock and clock enables differently than other types of ports. A clock port on an imported module must always be accompanied by a clock enable port, and vice versa. In other words, clock and clock enables must be defined as a pair, and exist as a pair in the imported module. This is true for both single synchronous clock and multiple independent clock designs.

SysgenBlockDescriptor provides a method called addClkCEPair that you can use to define clock, clock enable, and its associated clock period by using clock sub-system domain. The clock domain information is not required for synchronous single clock designs.

The first parameter defines the name of the clock port (as it appears in the module). The second parameter defines the name of the clock enable port (also as it appears in the module).

The port names of a clock and clock enable pair must follow the naming conventions provided below:

- The clock port must contain the substring `clk`.
- The clock enable must contain the substring `ce`.
- The strings containing the substrings `clk` and `ce` must be the same, such as: `my_clk_1` and `my_ce_1`.

The third parameter defines the rate relationship between the clock and the clock-enable port. The rate parameter should not be thought of as a Simulink® sample rate. Instead, this parameter tells Model Composer the relationship between the clock sample period, and the desired clock enable sample period. The rate parameter is an integer value that defines the ratio between the clock rate and the corresponding clock enable rate.

For multiple independent clock designs, the fourth and fifth optional parameters are mandatory.

The fourth parameter holds a "Boolean" value, and it defines whether clock and clock enable pair is tied to ground. If you set it to `true`, both clock and clock enable would be tied to ground during simulation. Setting it to false would activate clock and clock enable rate transitions.

The fifth parameter defines the clock period for the corresponding clock-clock enable pair. The 'clockDomain' property of the black box "SysgenPortDescriptor" must be used to set the clock periods for multiple independent clock designs.

Example:

```
rate_data = this_block.port('<port_name>').rate;
clkDomain_data = this_block.port(<port_name>).clockDomain;
this_block.addClkCEPair('clk',ce',rate_data, false, clkDomain_data);
```

HDL Co-Simulation

This topic describes how a mixed language/mixed flow design that includes Xilinx® HDL blocks, HDL modules, and a Simulink block design can be simulated in its entirety.

Model Composer simulates black boxes by automatically launching an HDL simulator, generating additional HDL as needed (analogous to an HDL test bench), compiling HDL, scheduling simulation events, and handling the exchange of data between the Simulink and the HDL simulator. This is called *HDL co-simulation*.

Configuring the HDL Simulator

Black box HDL can be co-simulated with Simulink® using the Model Composer interface to either the Vivado® simulator or the Questa simulation software from Model Technology, Inc.

Xilinx® Simulator

To use the Xilinx simulator for co-simulating the HDL associated with the black box, select **Vivado Simulator** as the option for the **Simulation mode** parameter on the black box. The model is then ready to be simulated and the HDL co-simulation takes place automatically.

Questa Simulator

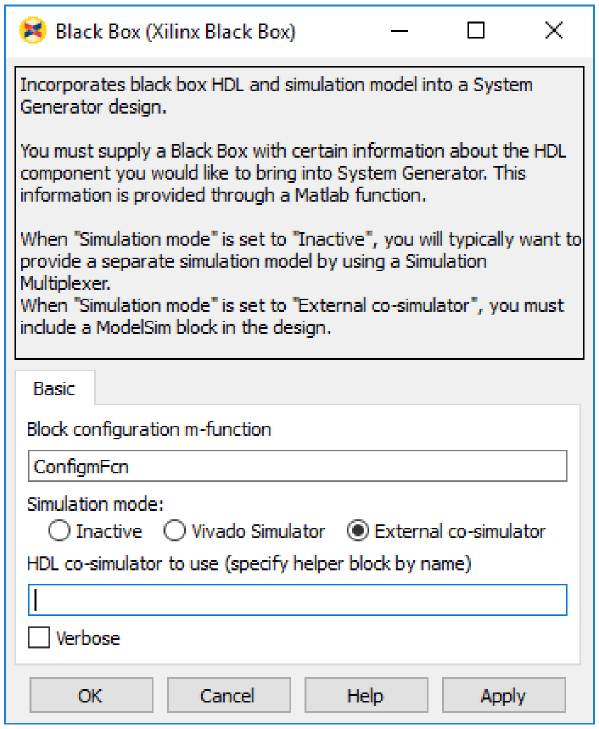
To use the Questa simulator by Model Technology, Inc., you must first add the Questa block that appears in the Tools library of the Xilinx HDL Blockset to your Simulink diagram.

Figure 130: **Questa Block**

For each black box that you wish to have co-simulated using the Questa simulator, you need to open its block parameterization dialog and set it to use the Questa session represented by the black box that was just added. You do this by making the following two settings:

1. Change the Simulation Mode field from Inactive to External co-simulator.
2. Enter the name of the Questa block (e.g., Questa) in the HDL co-simulator to use field.

Figure 131: **Black Box Parameters**



Black Box (Xilinx Black Box)

Incorporates black box HDL and simulation model into a System Generator design.

You must supply a Black Box with certain information about the HDL component you would like to bring into System Generator. This information is provided through a Matlab function.

When "Simulation mode" is set to "Inactive", you will typically want to provide a separate simulation model by using a Simulation Multiplexer.
When "Simulation mode" is set to "External co-simulator", you must include a ModelSim block in the design.

Basic

Block configuration m-function: ConfigmFcn

Simulation mode: Inactive, Vivado Simulator [✓ External co-simulator]

HDL co-simulator to use (specify helper block by name):

☐ Verbose

The block parameter dialog for the Questa block includes some parameters that you can use to control various options for the Questa session. See the block help page for details. The model is then ready to be simulated with these options, and the HDL co-simulation takes place automatically.

Co-Simulating Multiple Black Boxes

Model Composer allows many black boxes to share a common Questa co-simulation session. For example, many black boxes can be set to use the same Questa block. In this case, Model Composer automatically combines all black box HDL components into a single shared top-level co-simulation component, which is transparent to the user. However, only one Questa simulation license is needed to co-simulate several black boxes in the Simulink® simulation.

Multiple black boxes can also be co-simulated with the Vivado simulator by selecting **Vivado Simulator** as the option for **Simulation mode** on each black box.

Black Box Configuration Wizard

Model Composer provides a configuration wizard that makes it easy to associate a VHDL or Verilog module to a Black Box block. The Configuration Wizard parses the VHDL or Verilog module that you are trying to import, and automatically constructs a configuration M-function based on its findings. Then, it associates the configuration M-function it produces to the Black Box block in your model. Whether or not you can use the configuration M-function as is depends on the complexity of the HDL you are importing. Sometimes the configuration M-function must be customized by hand to specify details the configuration wizard misses. Details on the construction of the configuration M-function can be found in the Black Box Configuration M-Function topic.

Using the Configuration Wizard

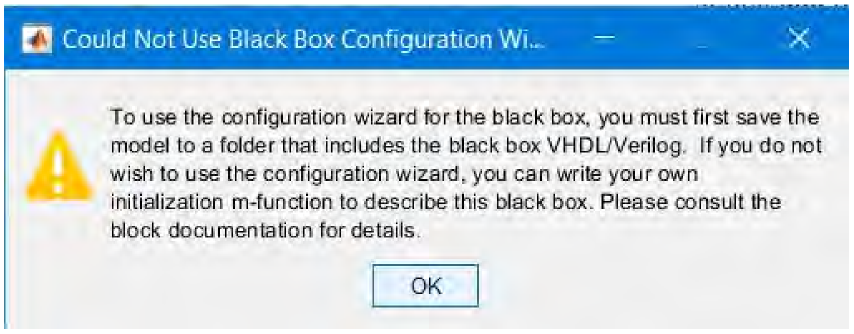
The Black Box Configuration Wizard opens automatically when a new black box block is added to a model.

***Note*:** Before running the Configuration Wizard, ensure the VHDL or Verilog you are importing meets the specified Black Box HDL Requirements and Restrictions.

For the Configuration Wizard to find your module, the model must be saved in the same directory as the module you are trying to import.

***Note*:** The wizard only searches for `.vhd` and `.v` files in the same directory as the model. If the wizard does not find any files it issues a warning and the black box is not automatically configured. The warning looks like the following:

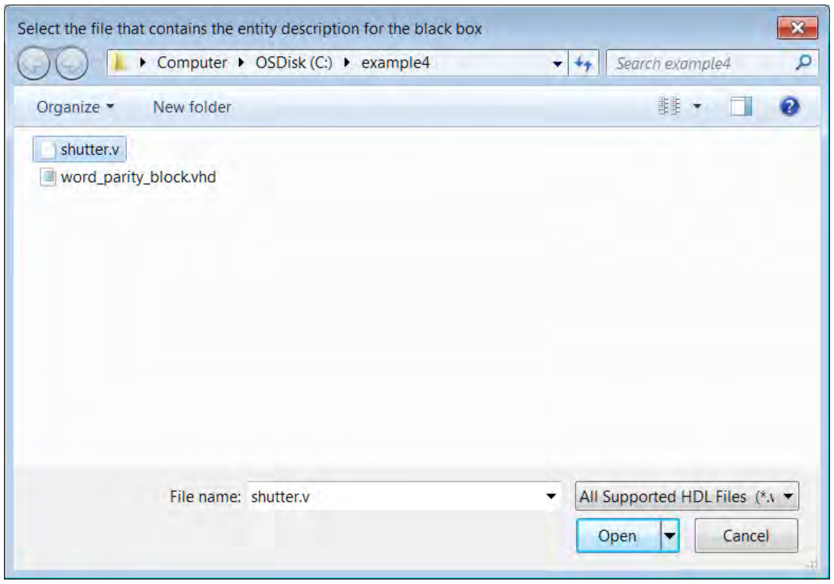
Figure 132: **Warning**



To use the configuration wizard for the black box, you must first save the model to a folder that includes the black box VHDL/Verilog. If you do not wish to use the configuration wizard, you can write your own initialization m-function to describe this black box. Please consult the block documentation for details.

After searching the model's directory for `.vhd` and `.v` files, the Configuration Wizard opens a new window that lists the possible files that can be imported. An example screenshot is shown below:

Figure 133: **Files to Import**


Select the file that contains the entity description for the black box

Computer ▸ OSDisk (C:) ▸ example4

shutter.v
word_parity_block.vhd

File name: shutter.v

All Supported HDL Files (*.v

You can select the file you would like to import by selecting the file, and then pressing the **Open** button. At this point, the configuration wizard generates a configuration M-function, and associates it with the black box block.

Note: The configuration M-function is saved in the model's directory as `<module>_config.m`, where `<module>` is the name of the module that you are importing.

Configuration Wizard Fine Points

The configuration wizard automatically extracts certain information from the imported module when it is run, but some things must be specified by hand. These things are described below:

Note: The configuration function is annotated with comments that instruct you where to make these changes.

- If your model has a combinational path, you must call the tagAsCombinational method of the block's SysgenBlockDescriptor object. A multiple independent hardware clock design will not support a combinational path.
- The Configuration Wizard only knows about the top-level entity that is being imported. There are typically other files that go along with this entity. These files must be added manually in the configuration M-function by invoking the addFile method for each additional file.
- The Configuration Wizard automatically creates either a synchronous single clock black box descriptor or an asynchronous multiple clock black box descriptor.
    - In the case of single-rate black box, every port on the black box runs at the same rate. In most cases, this is acceptable. You may want to explicitly set port rates, which can result in a faster simulation time.

- ◦ In the case of a multiple clock black box, the input port rate must be derived from the "source clock subsystem" and the output port rate must be set based on the "destination clock subsystem". In some cases, you may want to explicitly set port rates for a required configuration.

Compilation Types for HDL Library designs

There are different ways in which Model Composer can compile your design into an equivalent, often lower-level, representation. The way in which a design is compiled depends on settings in the System Generator Token dialog box. The support of different compilation types provides you the freedom to choose a suitable representation for your design's environment. For example, an HDL Netlist or IP catalog is an appropriate target if your design is used as a component in a larger system.

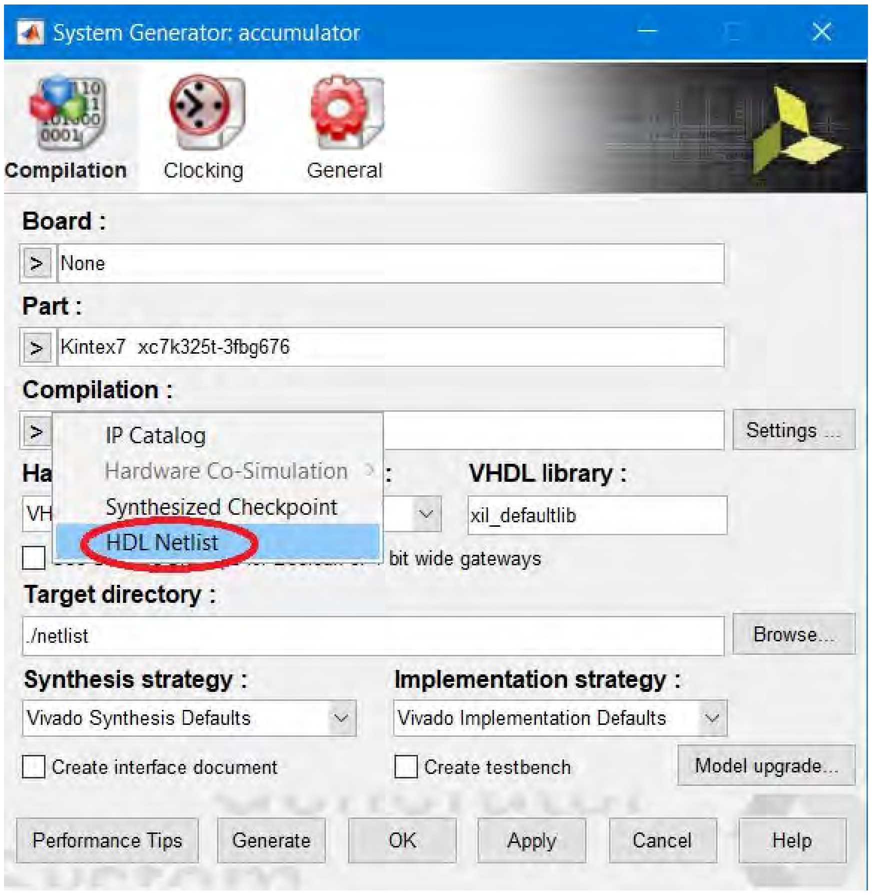
HDL Netlist Compilation	Describes how to generate HDL files that implement the design.
Hardware Co-Simulation Compilation	Describes how Model Composer can be configured to compile your design into FPGA hardware that can be used by Simulink® and Questa.
IP Catalog Compilation	Describes how to package a Model Composer design as an IP core that can be added to the Vivado® IP catalog for use in another design. Model Composer uses the IP catalog compilation type as the default generation target.
Synthesized Checkpoint Compilation	Describes how to generate a synthesized checkpoint file (`synth_1.dcp`) that can be used in a Vivado integrated design environment (IDE) project.

HDL Netlist Compilation

The **HDL Netlist** compilation type produces HDL files that implement the design. More details regarding the HDL Netlist compilation flow can be found in the Compilation Results section.

As shown below, you may select **HDL Netlist** compilation by left-clicking the **Compilation** submenu control on the System Generator token dialog box, and selecting the **HDL Netlist** target.

Figure 134: **HDL Netlist**



System Generator: accumulator

Compilation | Clocking | General

Board : None

Part : Kintex7 xc7k325t-3fbg676

Compilation : HDL Netlist (menu: IP Catalog, Hardware Co-Simulation, Synthesized Checkpoint, HDL Netlist)

VHDL library : xil_defaultlib

Target directory : ./netlist

Synthesis strategy : Vivado Synthesis Defaults

Implementation strategy : Vivado Implementation Defaults

☐ Create interface document ☐ Create testbench

The **Board** and **Part** fields allow you to specify the board or part for which you are targeting the **HDL Netlist** compilation. When you select a **Board**, the **Part** field automatically displays the name of the Xilinx® device on the selected **Board**, and this part name cannot be changed.

The HDL Netlist compilation can be performed for any of the boards or parts your Vivado tools support. In addition to accessing the Xilinx development boards installed as part of your Vivado installation, you can also specify Partner boards or custom boards (see Specifying Board Support in Model Composer HDL Blockset).

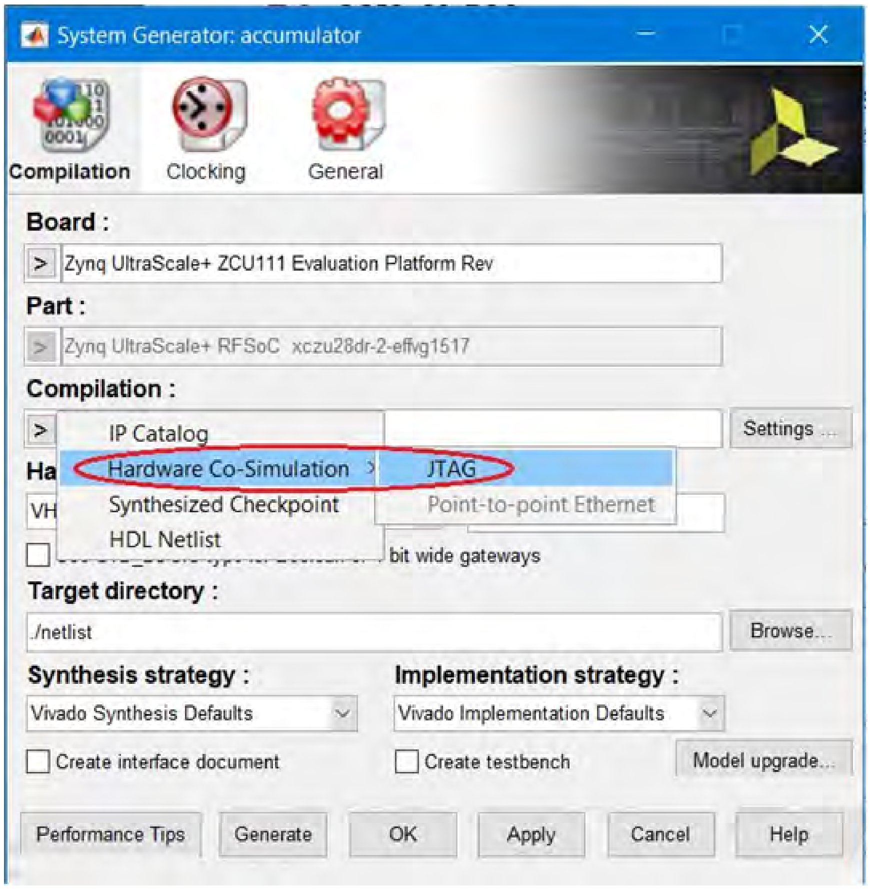
The files generated as part of an HDL Netlist compilation are placed in an `hdl_netlist` subdirectory under the directory you specified in the **Target directory** field. These files are described in the Compilation Results section.

Hardware Co-Simulation Compilation

Model Composer can compile designs into FPGA hardware that can be used in the loop with Simulink® simulations. This capability is discussed in the topic Using Hardware Co-Simulation.

As shown below, you may select **Hardware Co-Simulation** compilation by left-clicking the **Compilation** submenu control on the System Generator token dialog box, and selecting the **Hardware Co-Simulation** target.

Figure 135: **Hardware Co-Simulation**


System Generator: accumulator

Board : Zynq UltraScale+ ZCU111 Evaluation Platform Rev

Part : Zynq UltraScale+ RFSoC xczu28dr-2-effvg1517

Compilation : Hardware Co-Simulation > JTAG

Target directory : ./netlist

Synthesis strategy : Vivado Synthesis Defaults

Implementation strategy : Vivado Implementation Defaults

The **Board** fields allows you to specify the development board you are targeting when you are performing the Hardware Co-Simulation compilation. You can only select a Board for Hardware Co-Simulation compilation - you cannot select a Part. When you select a **Board**, the **Part** field automatically displays the name of the Xilinx® device on the selected **Board**, and this part name cannot be changed.

JTAG Hardware Co-Simulation is supported for all Xilinx development boards.

The Simulink library (`<design_name>_hwcosim_lib.slx`) generated as part of a Hardware Co-Simulation compilation is placed in the directory you specified in the **Target directory** field. This library, and the hardware co-simulation block stored in the library, are described in Hardware Co-Simulation Blocks.

IP Catalog Compilation

Model Composer uses the **IP Catalog** compilation type as the default generation target.

The IP Catalog compilation target allows you to package your Model Composer design into an IP module that can be included in the Vivado IP catalog. From there, the generated IP can be instantiated into another Vivado user design as a submodule.

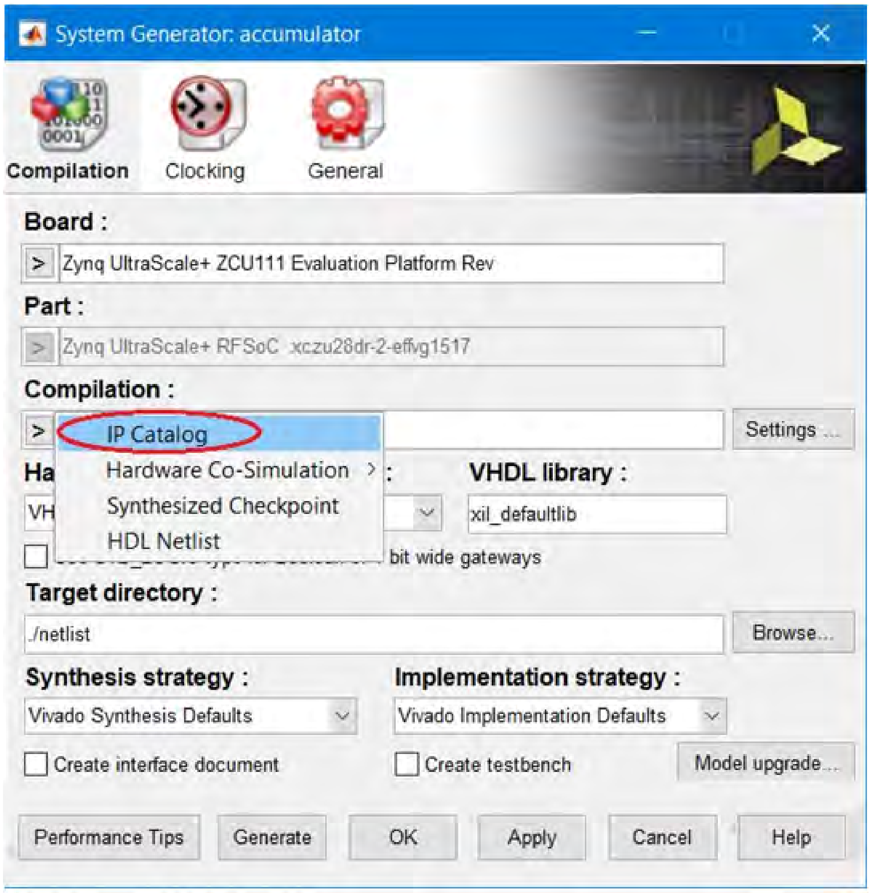
Model Composer first generates an HDL NetList based on the block design. If there are Vivado IP modules in the design, all the necessary IP files are copied into a subfolder named `IP`. Finally, all the RTL design files and Vivado IP design files are included into a ZIP file that is placed in a subfolder named `ip_catalog`.

The IP Catalog Flow

In a Model Composer design, double-click the System Generator token.

As shown below, under **Compilation**, click the **>** button, then select **IP Catalog**.

Figure 136: **IP Catalog**


System Generator: accumulator

Board : Zynq UltraScale+ ZCU111 Evaluation Platform Rev

Part : Zynq UltraScale+ RFSoC xczu28dr-2-ffvg1517

Compilation : IP Catalog

VHDL library : xil_defaultlib

Target directory : ./netlist

Synthesis strategy : Vivado Synthesis Defaults

Implementation strategy : Vivado Implementation Defaults

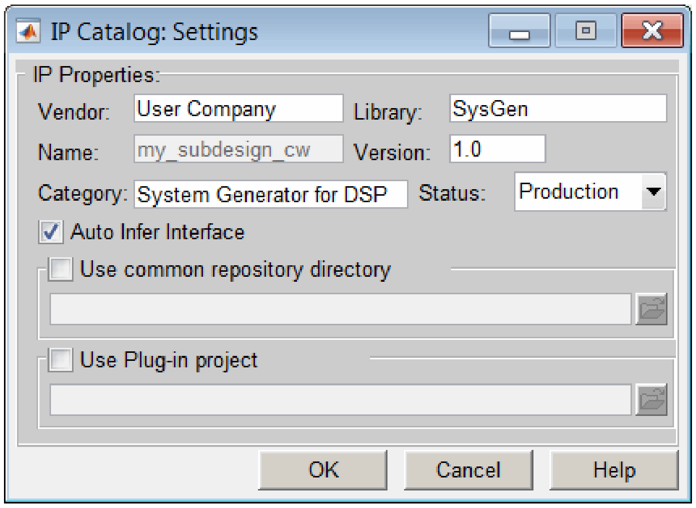
The **Board** and **Part** fields allow you to specify the board or part for which you are targeting the **IP Catalog** compilation. When you select a **Board**, the **Part** field automatically displays the name of the Xilinx device on the selected **Board**, and this part name cannot be changed.

The **IP Catalog** compilation can be performed for any of the boards or parts your Vivado tools support. In addition to accessing the Xilinx development boards installed as part of your Vivado installation, you can also specify Partner boards or custom boards (see Specifying Board Support in Model Composer HDL Blockset).

The **Target directory** field allows you to specify the location of the generated files.

The **Settings** button activates and when you click on it, a dialog box appears as shown below, allowing you to enter information about the module that will appear in the Vivado IP catalog.

Figure 137: **IP Catalog Settings**

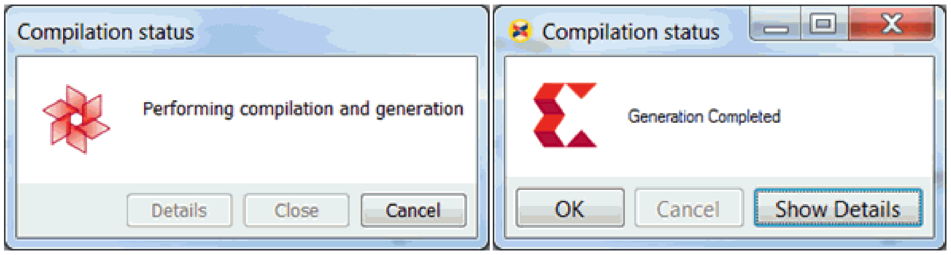


The **Use common repository directory** field allows you to specify a directory referred to as the Common Repository. In an IP catalog compilation, the IP created is copied over to this location. If a Vivado user adds this Path as User Repository in the Vivado project's IP Settings, then all IPs that a Model Composer user has placed in this Common Repository will automatically be picked up by Vivado and can be used either in an IP integrator or an RTL flow.

The **Use Plug-in project** field is used to specify a Vivado project containing an IP integrator Block Diagram (BD) that has been imported into Model Composer. For an example of a procedure that will need to have a Vivado project specified in this field, see Tailor Fitting a Platform Based Accelerator Design in Model Composer.

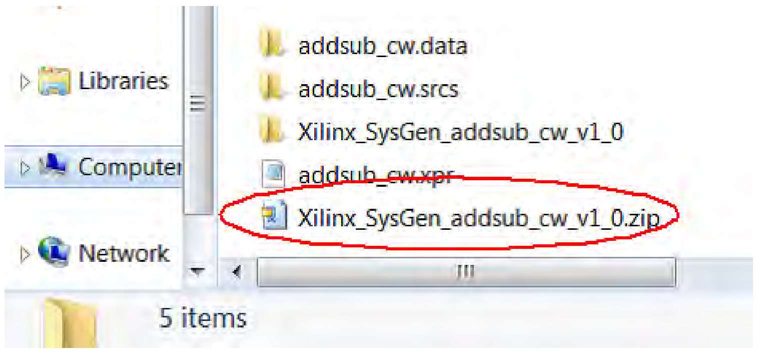
Once you click the **Generate** button, the IP catalog flow starts. As shown below, **Compilation status** windows pop up and indicate the progress of the flow. Once the IP Catalog flow is finished, it will indicate **Generation Completed**. You can then click **Show Details**, to get more detailed information.

Figure 138: **Compilation Status**



Navigate to the specified Target directory, to find a folder named `ip_catalog`. This folder contains all the necessary files to form an IP from your Model Composer design. The ZIP file, circled below, contains all the files required to include the Model Composer design as IP in the Vivado IP catalog.

Figure 139: **ZIP File**



Using AXI4 Interfaces

Selecting the **Auto Infer Interface** option in the IP Catalog: Settings dialog box ensures AXI4 interfaces are automatically inferred from the design Gateway In and Gateway Out ports. The **Auto Infer Interface** option groups signals into AXI4-Stream, AXI4-Lite and AXI4 interfaces based on the port names.

The **Auto Infer Interface** option will infer interfaces based on the following criteria:

- The Gateway In and Gateway Out port name suffix must exactly match the signal names in the AXI4 interface standard.
- The design must contain the minimum number of signals to be considered a valid AXI4 interface.

For example, if a design has two Gateway In ports named PortName_tdata and PortName_tvalid, and also a Gateway Out port named PortName_tready, the **Auto Infer Interface** option infers these three ports into a single AXI4-Stream port named PortName. In this example.

- The port name suffixes are exact matches for the signals in an AXI4-Stream interface (TDATA, TREADY and TVALID).

- These three signals are the minimum signals required for an AXI4-Stream interface.

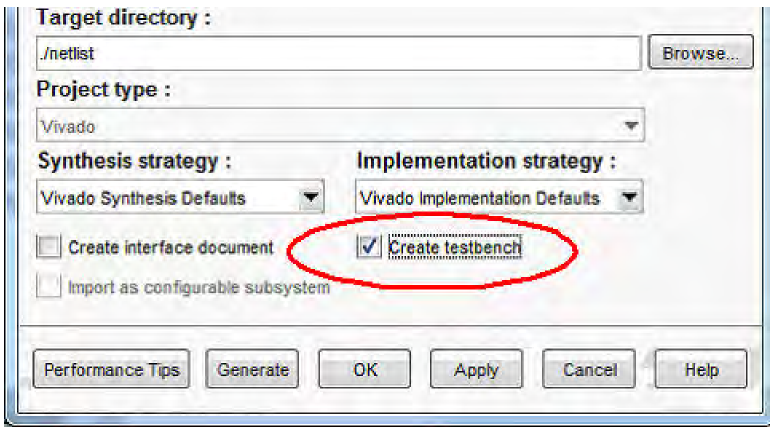
If optional AXI4 sideband signals are present, for example the TUSER signal is optional in the AXI4-Stream standard, and they are named using the same naming convention (for example, PortName_tuser) they will be grouped into the same AXI4 Interface.

For more details on AXI4 interfaces, AXI4 interface signals names and the minimum required signals for an AXI4 interface, refer to the document *Vivado Design Suite: AXI Reference Guide* (UG1037).

Including a Testbench with the IP Module

To verify the functionality of the newly generated IP, it is important to include a test bench. As shown below, if you check **Create testbench**, a test bench is automatically created when you click the **Generate** button.

Figure 140: **Create Testbench**

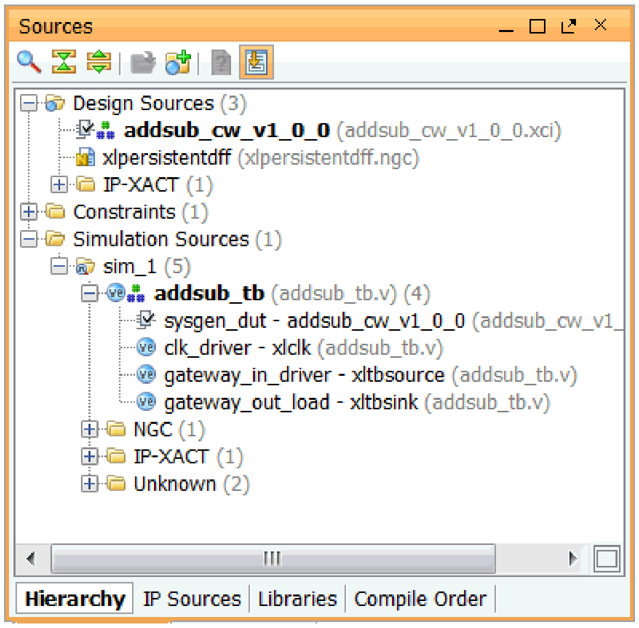

Target directory : ./netlist
Project type : Vivado
Synthesis strategy : Vivado Synthesis Defaults
Implementation strategy : Vivado Implementation Defaults
[ ] Create interface document
[✓] Create testbench
[ ] Import as configurable subsystem

As shown below, when you include a test bench, you can verify the IP functionality by adding three more steps to the flow.

- **Step 1:** Add the new IP to the Vivado IP catalog. Refer to the document *Vivado Design Suite User Guide: Designing with IP* (UG896).
- **Step 2:** Create a new Vivado IDE project and add the IP as the top-level source.
- **Step 3:** Run simulation, synthesis and implementation to verify the functionality of the generated IP.

The following figure shows an open Vivado IDE project with the newly created IP as the top-level source.

Figure 141: **New IP**

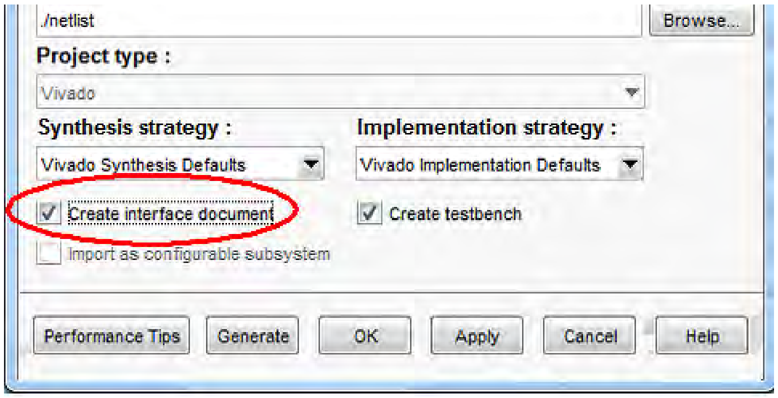

Sources

- Design Sources (3)
  - addsub_cw_v1_0_0 (addsub_cw_v1_0_0.xci)
  - xlpersistentdff (xlpersistentdff.ngc)
  - IP-XACT (1)
- Constraints (1)
- Simulation Sources (1)
  - sim_1 (5)
    - addsub_tb (addsub_tb.v) (4)
      - sysgen_dut - addsub_cw_v1_0_0 (addsub_cw_v1_
      - clk_driver - xlclk (addsub_tb.v)
      - gateway_in_driver - xltbsource (addsub_tb.v)
      - gateway_out_load - xltbsink (addsub_tb.v)
    - NGC (1)
    - IP-XACT (1)
    - Unknown (2)

Adding an Interface Document to the IP Module

As shown below, check **Create interface document**, then click **Generate**, and Model Composer generates an interface document for the IP and packages this HTML document with the IP.

Figure 142: **Create Interface Document**


./netlist

Project type : Vivado

Synthesis strategy : Vivado Synthesis Defaults

Implementation strategy : Vivado Implementation Defaults

☑ Create interface document

☑ Create testbench

☐ Import as configurable subsystem

You can find a new folder, `documentation`, under the `netlist` folder. Right-click the new IP in the Vivado IDE, and click **Product guide**, to open one HTML file with interface information about this IP.

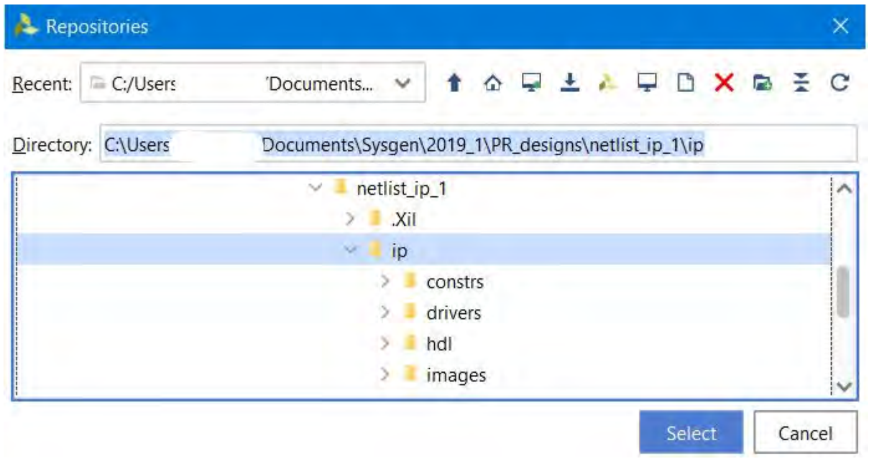
Adding the Generated IP to the Vivado IP Catalog

To use the IP generated from Model Composer, you need to create a new project, or open an existing project that targets the same device as specified in Model Composer for creating the IP.

Note: The IP is only accessible in this project. For each new project where you use this IP, you need to perform the same steps.

Select **IP Catalog** in the Project Manager, and right-click an empty area in IP Catalog window. Select **Add Repository**, and add the directory that contains your new IP.

Figure 143: **IP Catalog**


Repositories
Recent: C:/Users 'Documents...
Directory: C:\Users Documents\Sysgen\2019_1\PR_designs\netlist_ip_1\ip
netlist_ip_1
.Xil
ip
constrs
drivers
hdl
images
Select
Cancel

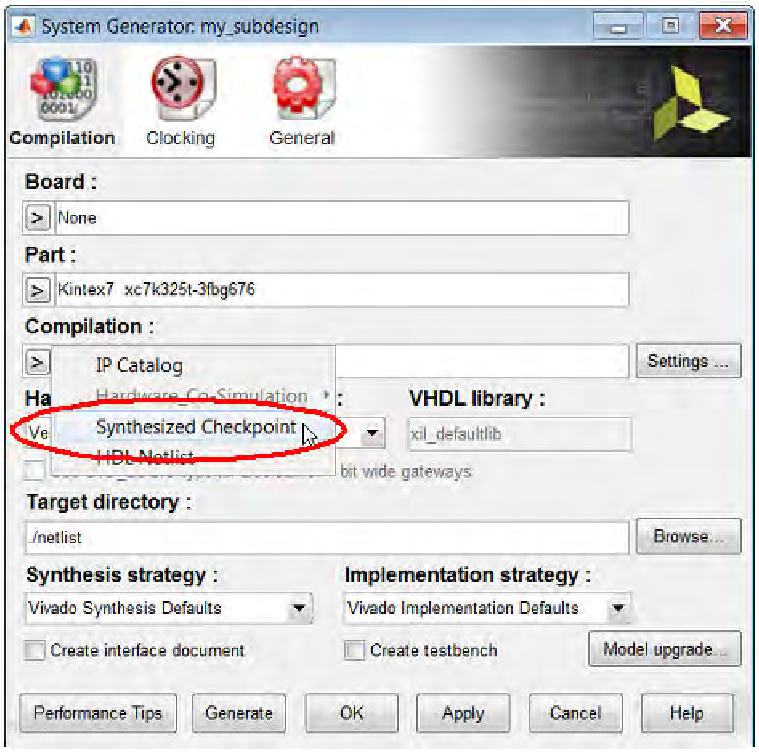
Once the IP is added to the IP catalog, you can include it in larger designs just as you would with any other IP in the IP catalog.

Synthesized Checkpoint Compilation

Vivado tools provide design checkpoint files (`.dcp`) as a mechanism to save and restore a design at key steps in the design flow. Checkpoints are merely a snapshot of a design at a specific point in the flow. A Synthesized Checkpoint is a checkpoint file that is created in the out-of-context (OOC) mode after a design has been successfully synthesized.

When you select the **Synthesized Checkpoint** compilation target (see figure below), a synthesized checkpoint target file named `<design_name>.dcp` is created, and placed in the **Target directory**. You can then use this `<design_name>.dcp` file in any Vivado IDE project.

Figure 144: **Synthesized Checkpoint**


System Generator: my_subdesign

Board : None

Part : Kintex7 xc7k325t-3fbg676

Compilation :

VHDL library : xil_defaultlib

bit wide gateways

Target directory : ./netlist

Synthesis strategy : Vivado Synthesis Defaults

Implementation strategy : Vivado Implementation Defaults

Create interface document

Create testbench

The **Board** and **Part** fields allow you to specify the board or part for which you are targeting the Synthesized Checkpoint compilation. When you select a **Board**, the **Part** field automatically displays the name of the Xilinx device on the selected **Board**. This part name cannot be changed.

The Synthesized Checkpoint compilation can be performed for any of the boards or parts your Vivado tools support. In addition to accessing the Xilinx development boards installed as part of your Vivado installation, you can also specify partner boards or custom boards (see Specifying Board Support in Model Composer HDL Blockset).

Creating Your Own Custom Compilation Target

Model Composer provides a custom compilation infrastructure to create your own custom compilation target. In addition to generating HDL from your Model Composer design, you can also create a compilation target plug-in that automates steps both before and after the HDL is generated. Details about how to create a custom compilation target can be found in the topic Creating Custom Compilation Targets topic.

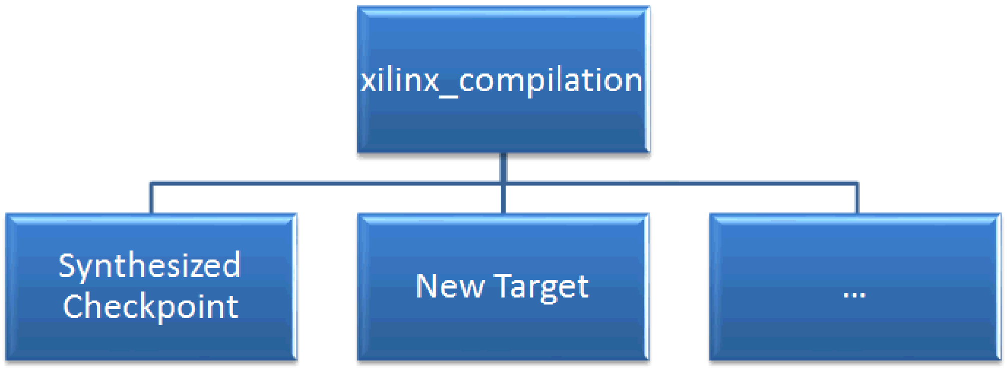
Creating Custom Compilation Targets

Model Composer provides a custom compilation infrastructure that allows you to create your own custom compilation targets. In addition to generating HDL from your Model Composer design, you can also create a compilation target plug-in that automates steps both before and after the Vivado® integrated design environment (IDE) project is created. In order to create a custom compilation target, you need to be familiar with the object-oriented programming concepts in the MATLAB® environment.

xilinx_compilation Base Class

The custom compilation infrastructure provides a base class named `xilinx_compilation`. From this base class, you can then create a subclass, and use its properties and override the member functions to implement your own functionality.

Figure 145: **Base Class**



Creating a New Compilation Target

The following general procedure outlines how to create a new compilation target, and is followed by more specific examples.

Running the Helper Function

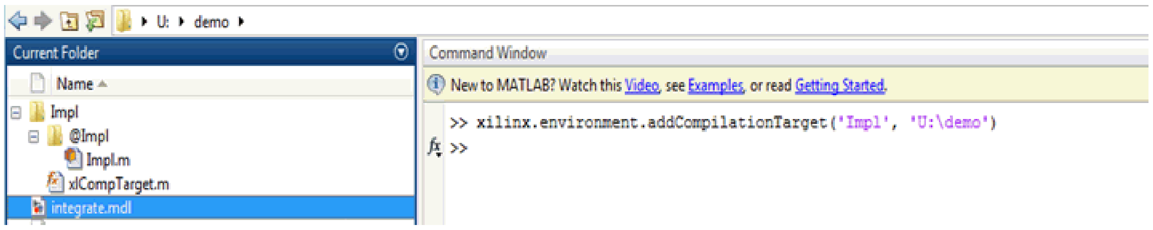
Create a new custom compilation target by running the following helper function.

```
xilinx.environment.addCompilationTarget(target_name, directory_name)
```

For example, consider the following command:

```
xilinx.environment.addCompilationTarget('Impl', 'U:\demo')
```

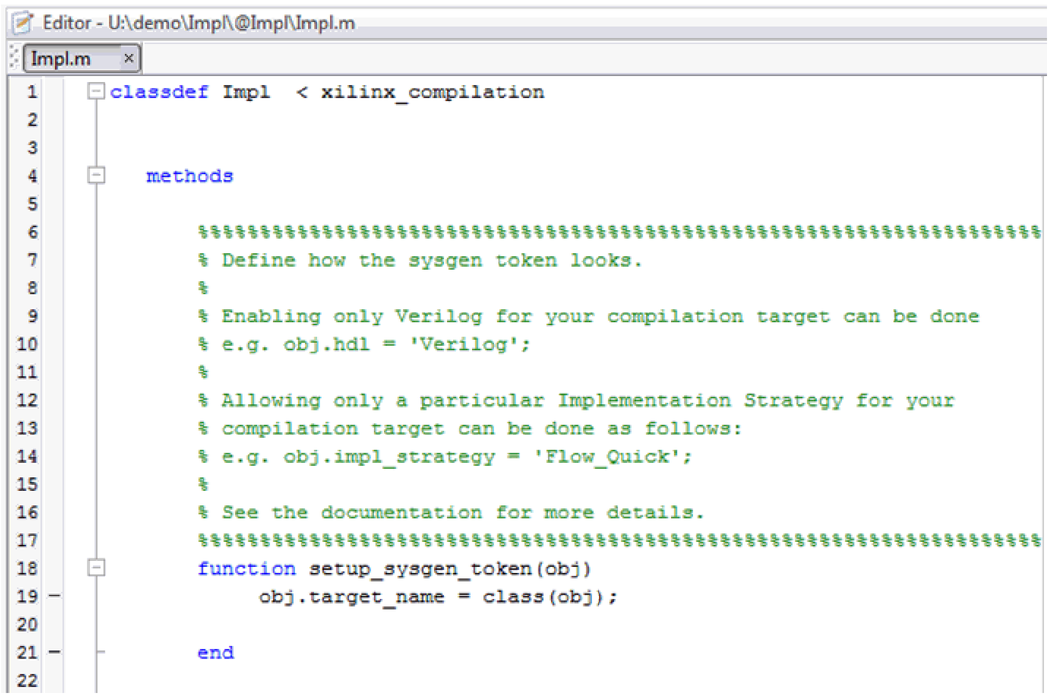
Figure 146: **Helper Function**



```
>> xilinx.environment.addCompilationTarget('Impl', 'U:\demo')
>>
```

When you enter this command in the MATLAB Command Window as shown above, the following happens:

1. A folder is created named `Impl/@Impl` in `U:\demo`.
2. Inside the folder, a template class file `Impl` is created (`Impl.m`), which is derived from the base class `xilinx_compilation`. At this point, if no modifications are made to the file, the newly created `Impl` compilation target acts the same as the **HDL Netlist** compilation target. The content of the `Impl.m` file is shown in the following figure.



```
classdef Impl  < xilinx_compilation


    methods

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Define how the sysgen token looks.
        %
        % Enabling only Verilog for your compilation target can be done
        % e.g. obj.hdl = 'Verilog';
        %
        % Allowing only a particular Implementation Strategy for your
        % compilation target can be done as follows:
        % e.g. obj.impl_strategy = 'Flow_Quick';
        %
        % See the documentation for more details.
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        function setup_sysgen_token(obj)
            obj.target_name = class(obj);

        end

```

3. The helper function then adds `U:\demo\Impl` to the MATLAB path, so that the new class `Impl` can be discovered by MATLAB.

***Note*:** Be aware that the `target_name` cannot contain spaces. After the class is created, you can add spaces to the `target_name` property of the class.

Modifying a Compilation Target

If modifications are made to a class file for a compilation target, you are required to call the following helper function. This helper function ensures that Model Composer detects the new class definition.

```
>> xilinx.environment.rehashCompilationTarget
```

Adding an Existing Compilation Target

You must add the path that contains the folder with the custom compilation target. As shown below, you can use the `addpath` functionality provided by MATLAB® to do this:

```
>>addpath('U:\demo\Impl');
```

When you use `addpath`, you must provide the absolute path, not the relative path.

Saving a Custom Compilation Target

You can use the `savepath` functionality in MATLAB® to save the custom compilation target. To do the save, you may need write permission to the MATLAB installation area.

Removing a Custom Compilation Target

To remove the custom compilation target, remove the path to the target from the MATLAB® Search Path.

Base Class Properties and APIs

The xilinx_compilation base class resides in the following location:

```
<Vivado Install Path>/scripts/sysgen/matlab/@xilinx_compilation
```

System Generator Token-Related Properties and APIs

setup_sysgen_token()

This function is called to populate the System Generator token information by the Custom Compilation Infrastructure. You can use any of the following functions related to the System Generator token to set how the token looks by default when the custom target is selected. The fields, their default values and the field enablement/disablement can be set by the following System Generator token application programming interface (API) functions.

add_part(family, device, speed, package, temperature)

An example of an explicit command is `add_part('Kintex7', 'xc7k325t', '-1' , 'fbg676','')`. If the part-related APIs are not used, the end user can select any device that he wants to choose from the list.

string target_name

This is a required field that has to be set in the `setup_sysgen_token()` function.

string hdl

The default value is an empty string. Valid options are `verilog` or `vhdl`. Once a value is set to this field, this field will be disabled for further user selection.

string synth_strategy

The default value is an empty string. Once a value is set to this field, this field will be disabled for further user selection. If this API is used, the user has to make sure that the specified strategy exists. Otherwise, it will result in an error.

string impl_strategy

The default value is an empty string. Once a value is set to this field, this field will be disabled for further user selection. If this API is used, the user has to make sure that the specified strategy exists. Otherwise, it will result in an error.

string create_tb

The default value is an empty string. The valid options are `on` or `off`. Once a value is set to this field, this field will be disabled for further user selection.

string create_iface_doc

The default value is an empty string. The valid options are `on` or `off`. Once a value is set to this field, this field will be disabled for further user selection.

Vivado Project-Related Properties

top_level_module

You can use this property to set the top-level name of their choice. This parameter accepts a MATLAB® string.

Vivado IDE Project Generation-Related Functions

pre_project_creation(design_info)

This function should be called before you create the Vivado® IDE project. Before the Model Composer Infrastructure creates the project, it has to know what files need to be added to the Vivado® IDE project, and what additional Tcl commands need to be run. There might be use-cases where the user wants to add some files to the project based on the top-level port interface of the Model Composer design. For this purpose, a structure that describes the port interface is passed into this function called `design_info`. `design_info` is described in detail in a later section.

post_project_creation(design_info)

This function should be called at the end of Vivado IDE project creation. This is the last function to be called after the Project Generation script is run. This is a useful function for things like error parsing, generating reports, and opening the Vivado IDE project. A structure which describes the port interface is passed into this function called `design_info`. `design_info` is described in detail in a later section.

add_tcl_command(string)

This function adds the additional Tcl commands as a string. These Tcl commands will be issued after the Vivado IDE project is created. Use this command to create a bitstream once project creation occurs. The Tcl command can also be used to source a particular Tcl file. The commands are executed in the order in which they are received.

add_file(string)

This function adds user-defined files to the Vivado IDE project. This application programming interface (API) function can also be used to add XDC constraint files to the Vivado IDE project. You should make sure that the order in which add_file is called, is hierarchical in nature. The top-module file must be added last.

run_synthesis()

This function runs synthesis in the Vivado IDE project.

run_implementation()

This function runs implementation in the Vivado IDE project.

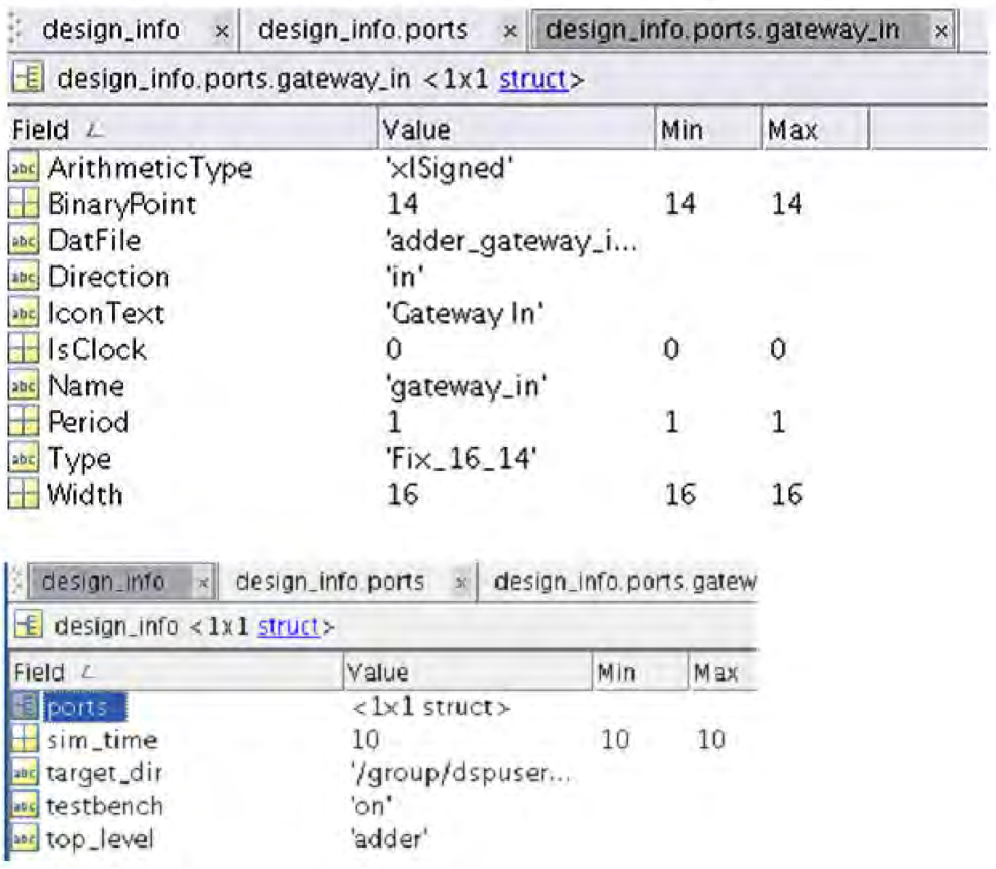
generate_bitstream()

This function generates a bitstream in the Vivado IDE project.

Design Info

`design_info` is a MATLAB® struct and its contents are shown below:

Figure 147: **Design Info**



design_info
design_info.ports
design_info.ports.gateway_in
design_info.ports.gateway_in <1x1 struct>

Field	Value	Min	Max
ArithmeticType	'xlSigned'		
BinaryPoint	14	14	14
DatFile	'adder_gateway_i...		
Direction	'in'		
IconText	'Gateway In'		
IsClock	0	0	0
Name	'gateway_in'		
Period	1	1	1
Type	'Fix_16_14'		
Width	16	16	16

Field	Value	Min	Max
ports	<1x1 struct>		
sim_time	10	10	10
target_dir	'/group/dspuser...		
testbench	'on'		
top_level	'adder'		

Examples of Creating Custom Compilation Targets

The following examples provide more detail on how you can create various kinds of customized targets.

Example 1: Creating an Implementation Target

1. Open a Model Composer model, then open the System Generator token. This populates the token with all the available compilation targets.
2. In the MATLAB® Command Window, modify the path as per your requirements, then enter the following command:

   ```
   xilinx.environment.addCompilationTarget('Impl', 'U:\demo')
   ```

   This provides a template derived class for you to edit.

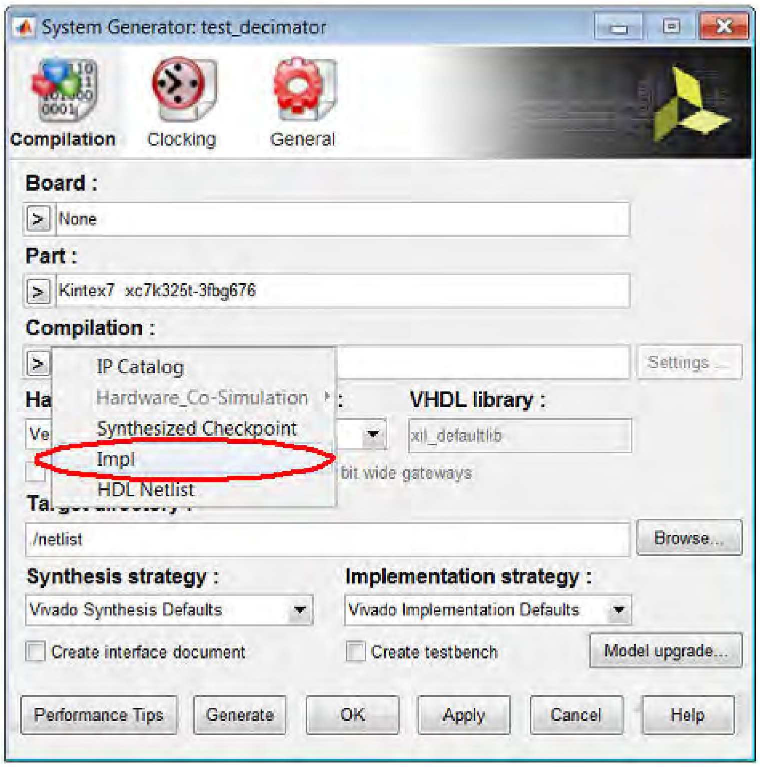
3. In the MATLAB Command Window, enter the following command:

```
xilinx.environment.rehashCompilationTarget
```

   This ensures that the new compilation target is picked up by the System Generator token.

4. Close and then re-open the System Generator token. You now see the compilation target, **Impl** on the token as shown below.

Figure 148: **Selecting Impl**



System Generator: test_decimator

Board : None

Part : Kintex7 xc7k325t-3fbg676

Compilation :

Target directory : /netlist

Synthesis strategy : Vivado Synthesis Defaults

Implementation strategy : Vivado Implementation Defaults

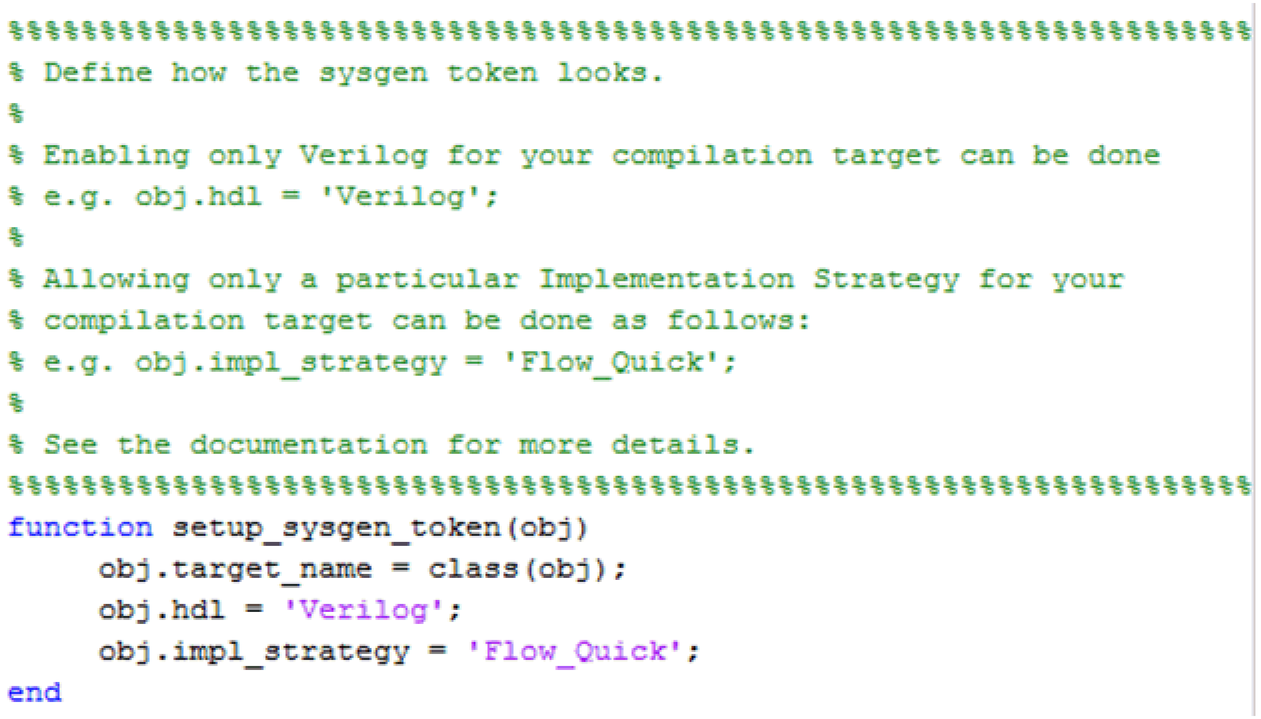
Create interface document

Create testbench

5. At this point, selecting **Impl** does not perform any customized operations on the System Generator token. It is equivalent to an HDL Netlist compilation target.

6. Open `U:\demo\Impl\@Impl\Impl.m` in the MATLAB Editor.

7. Populate the `setup_sysgen_token()` function as per the requirements. Using this approach, you can control how the System Generator token should look, including the enabled/disabled fields when the user-defined custom compilation is selected.



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Define how the sysgen token looks.
%
% Enabling only Verilog for your compilation target can be done
% e.g. obj.hdl = 'Verilog';
%
% Allowing only a particular Implementation Strategy for your
% compilation target can be done as follows:
% e.g. obj.impl_strategy = 'Flow_Quick';
%
% See the documentation for more details.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function setup_sysgen_token(obj)
     obj.target_name = class(obj);
     obj.hdl = 'Verilog';
     obj.impl_strategy = 'Flow_Quick';
end
```

8. In the MATLAB Command Window, you should enter the following command:

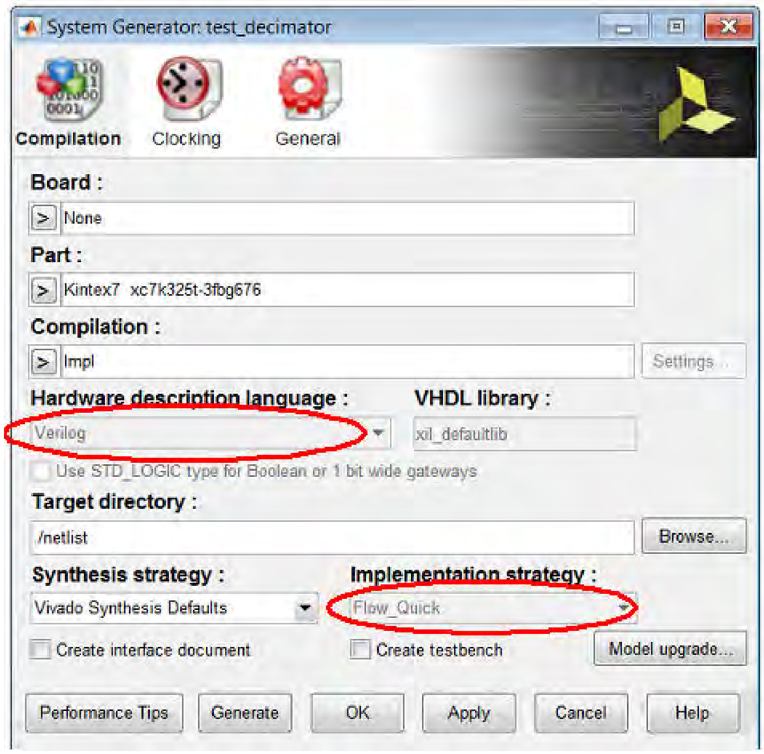
   ```
   xilinx.environment.rehashCompilationTarget
   ```

   This ensures that the updated class definition of **Impl** is used.

9. Close and then re-open the System Generator token. Select **Impl** from the list of Compilation targets.

10. The System Generator token appears as follows:

Figure 149: **Selecting Verilog, and Flow Quick**



Board : None

Part : Kintex7 xc7k325t-3fbg676

Compilation : Impl

Hardware description language : Verilog

VHDL library : xil_defaultlib

Use STD_LOGIC type for Boolean or 1 bit wide gateways

Target directory : ./netlist

Synthesis strategy : Vivado Synthesis Defaults

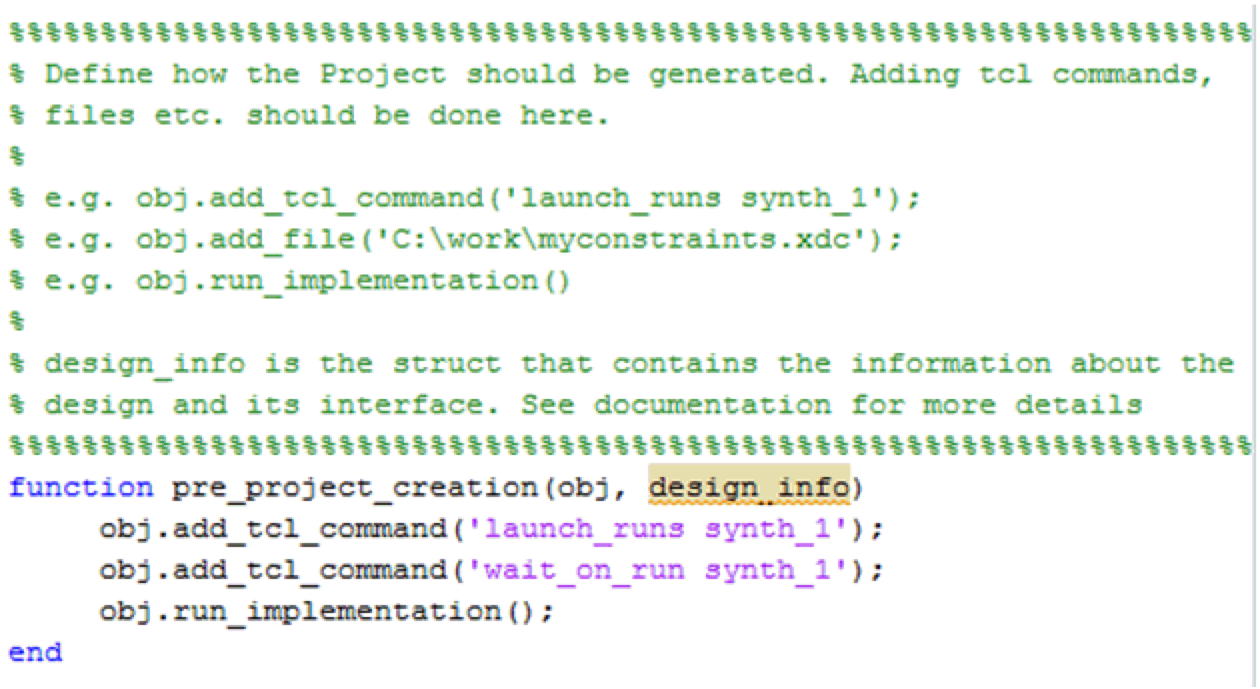
Implementation strategy : Flow_Quick

Create interface document

Create testbench

11. Observe that the **Hardware description language** field and the **Implementation strategy** field are fixed to what you set in the **Impl** class and are disabled for user modification.

12. All the user specified files and additional Tcl commands to be run are known before the Vivado® IDE project is created. The next step is to populate the `pre_project_creation()` function as indicated below:



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Define how the Project should be generated. Adding tcl commands,
% files etc. should be done here.
%
% e.g. obj.add_tcl_command('launch_runs synth_1');
% e.g. obj.add_file('C:\work\myconstraints.xdc');
% e.g. obj.run_implementation()
%
% design_info is the struct that contains the information about the
% design and its interface. See documentation for more details
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function pre_project_creation(obj, design_info)
    obj.add_tcl_command('launch_runs synth_1');
    obj.add_tcl_command('wait_on_run synth_1');
    obj.run_implementation();
end
```

13. In the MATLAB Command Window, enter the following command:

```
xilinx.environment.rehashCompilationTarget
```

    This ensures that the updated class definition of Impl is used.

14. Close and then re-open the System Generator token. Select **Impl** from the list of Compilation targets.

15. Click **Generate**. Once the process is finished, you can see the implementation results by opening up the Vivado IDE project.

Example 2: Creating a Bitstream Target

1. Open a Model Composer design.

2. In the MATLAB command Window, modify the path as per your requirements, similar to the first example, and then enter the following command:

```
xilinx.environment.addCompilationTarget('Bitstream', '.')
```

    This provides a template derived class for the users to edit. The last field corresponds to the directory which contains the `board.xml` file.

3. In the MATLAB Command Window, enter the following command:

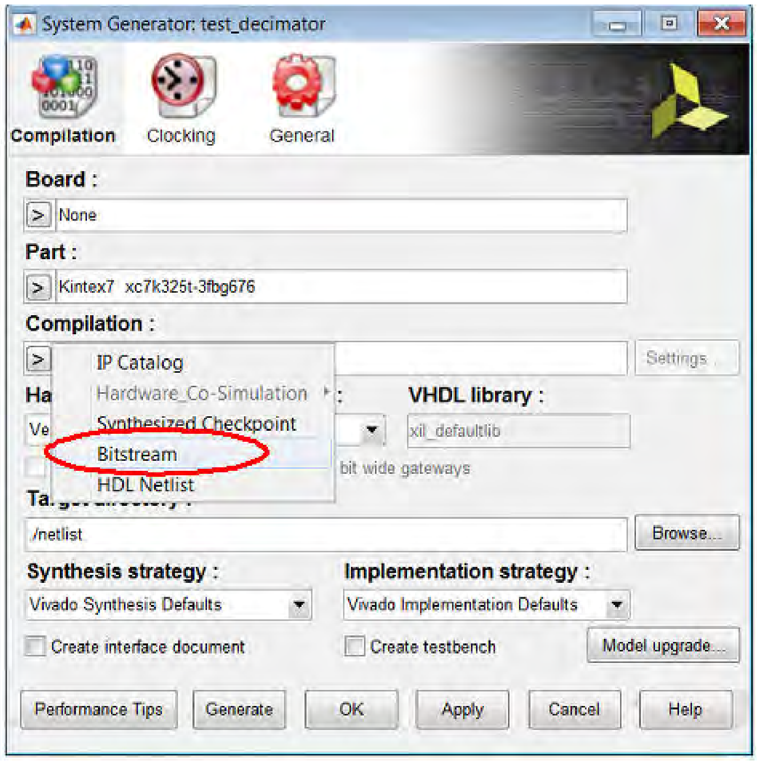
```
xilinx.environment.rehashCompilationTarget
```

    This will ensure that the new compilation target is picked up by the System Generator token

4. Close and then re-open the System Generator token.

5. You will now see the compilation target **Bitstream** on the System Generator token as shown below.

Figure 150: **Bitstream**



System Generator: test_decimator

Board : None

Part : Kintex7 xc7k325t-3fbg676

VHDL library : xil_defaultlib

Target directory : ./netlist

Synthesis strategy : Vivado Synthesis Defaults

Implementation strategy : Vivado Implementation Defaults

6. Open the `Bitstream.m` created in the '`./Bitstream/@Bitstream/Bitstream.m`'.
7. Download the following two files:

   bitstream_example.xdc   top.v

8. Inside the function `pre_project_creation()`, add the following lines to do the following:
   a. Set the board as a KC705 board.
   b. Add a new top-level file (`top.v`) to use the differential clock ports of KC705.
   c. Add a new XDC file to give the location constraints for the clock, dip, and led ports.
   d. Set the newly added module `top` as the top.
   e. Run synthesis.
   f. Run implementation.
   g. Generate bitstream.

After you save the files to a location on your computer, you should give the full path to the files in the add_file API as per your path.

```
add_tcl_command(obj, 'set_property board xilinx.com:kintex7:kc705:1.1
[current_project]');
add_file(obj,
'/group/dspusers-xsj/umangp/rel/2013.4/cust_comp_test/
bitstream_example.xdc');
add_file(obj, '/group/dspusers-xsj/umangp/rel/2013.4/cust_comp_test/
top.v');
obj.top_level_module = 'top';
run_synthesis(obj);
run_implementation(obj);
generate_bitstream(obj);
```

9. In the MATLAB Command Window, enter the following command:

   ```
   xilinx.environment.rehashCompilationTarget
   ```

   This ensures that the new compilation target is picked up by the System Generator token

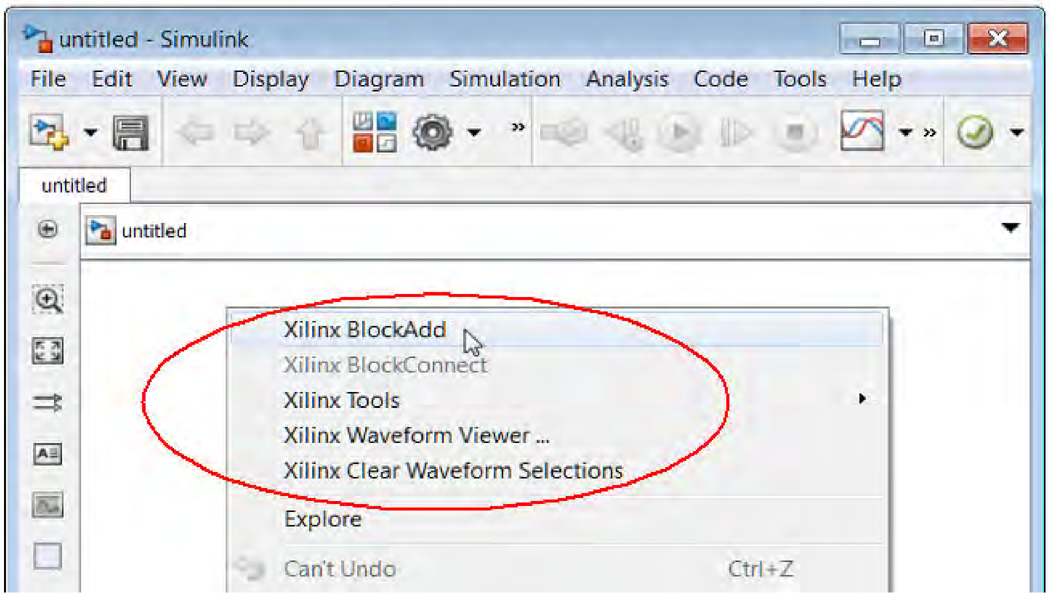
10. Close and then re-open the System Generator token.
11. Select the **Bitstream** compilation target.
12. Click the **Generate** button.
13. After the generation is complete, you can find the bit file in the following directory:

    ```
    ./<Target_directory>/Bitstream/bitstream_example.runs/impl_1/top.bit
    ```

GUI Utilities for HDL Blocksets

Xilinx has added graphics commands to the Simulink® model popup menu that will help you rapidly create and analyze your Model Composer design. As shown below, you can access these commands by right-clicking the Simulink model canvas and selecting the appropriate Xilinx command:

Figure 151: **Xilinx Commands**


untitled - Simulink
File Edit View Display Diagram Simulation Analysis Code Tools Help
untitled
untitled
Xilinx BlockAdd
Xilinx BlockConnect
Xilinx Tools
Xilinx Waveform Viewer ...
Xilinx Clear Waveform Selections
Explore
Can't Undo Ctrl+Z

A detailed description of the additional Xilinx commands is provided below.

Xilinx BlockAdd	Facilitates the rapid addition of Xilinx HDL blocks (and a limited set of Simulink blocks) to a Simulink model.
Xilinx Tools > Save as blockAdd default	This feature allows you to pre-configure a block, then add multiple copies of the pre-configured block using the BlockAdd feature.
Xilinx BlockConnect	Facilitates the rapid connection of blocks in a Simulink model
Xilinx Tools > Terminate	Facilitates the rapid addition of Simulink terminator blocks on open output ports and/or Xilinx HDL Constant Blocks on open input ports.
Xilinx Waveform Viewer	The Xilinx Waveform Viewer displays a waveform diagram of selected signals in your Model Composer design. Waveforms can be displayed in the Waveform Viewer after running a Simulink simulation. Inputs and outputs of blocks in the Xilinx HDL Blockset can be displayed in the Waveform Viewer.
Xilinx Clear Waveform Selections	Deletes all of the waveforms currently displayed in the Waveform Viewer, and closes the Waveform Viewer.

Xilinx BlockAdd

Facilitates the rapid addition of Xilinx® HDL blocks (and a limited set of Simulink® blocks) to a Simulink model.

How to Invoke

Method 1

Right-click the Simulink canvas and select **Xilinx BlockAdd**.

Method 2

Execute the short cut Ctrl 1 (one).

Method 3

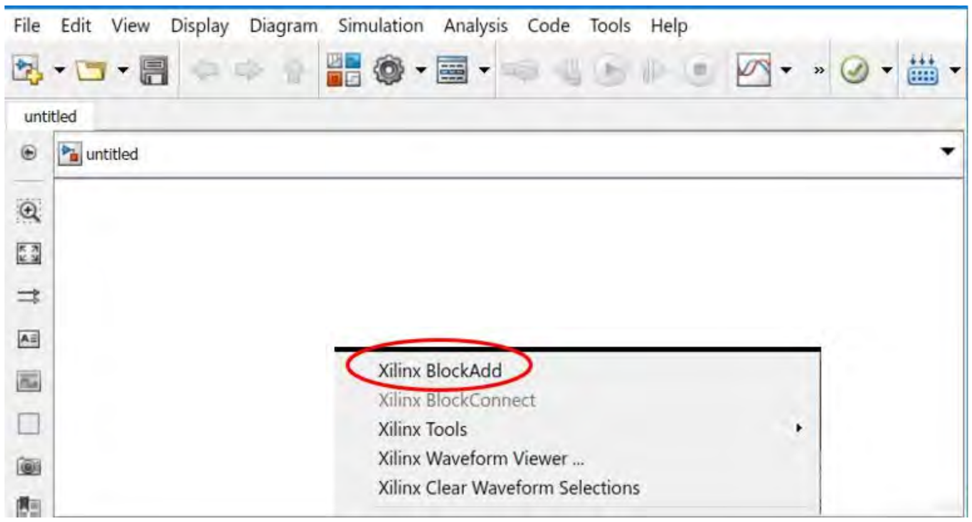
From the Simulink model pull down menu, select the following item:

Tools → Xilinx → BlockAdd

How to Use

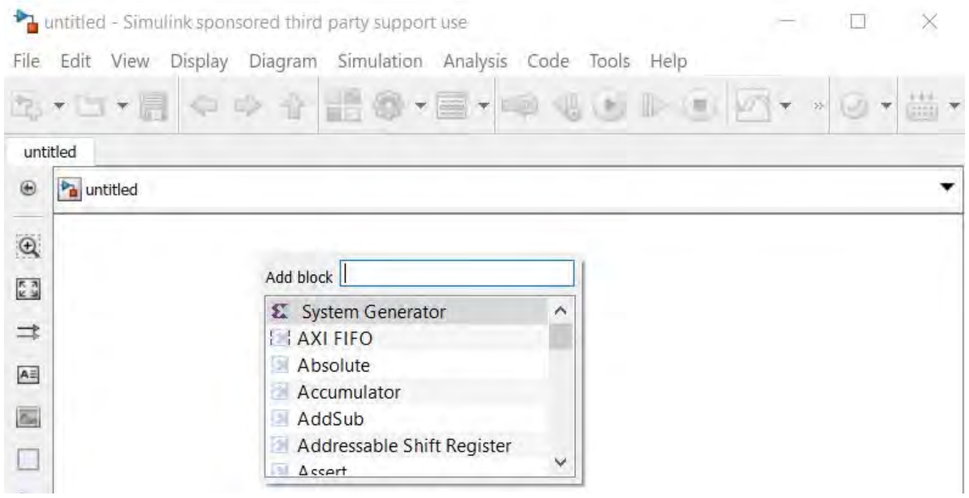
1. Right-click the Simulink canvas and select **Xilinx BlockAdd**.

Figure 152: **Xilinx BlockAdd**



2. Double-click **AddSub**.

Figure 153: **AddSub**



3. To add multiple copies of the same block, add a block, select the block, press **Ctrl + C**, then press **Ctrl + V** the required number of times.
4. To dismiss the Add block window, press **Esc**.

Xilinx Tools > Save as blockAdd default

This feature allows you to pre-configure a block, then add multiple copies of the pre-configured block using the BlockAdd feature.

How to Use

Assume you need to add multiple Gateway In blocks of type **Boolean** to a model.

1. Add one Gateway In block to the model.
2. Double click the **Gateway In** block, change the **Output type** to **Boolean** and click **OK**.
3. Select the modified **Gateway In** block, right-click and select **Xilinx Tools → Save as blockAdd default**.
4. Now, every time you add addition Gateway In blocks to the model using the BlockAdd feature, the block is of Output type Boolean.

How to Restore the Block Default

1. Select a block with pre-configured (changed) defaults.
2. Right-click and select **Xilinx Tools → Clear blockAdd defaults**.

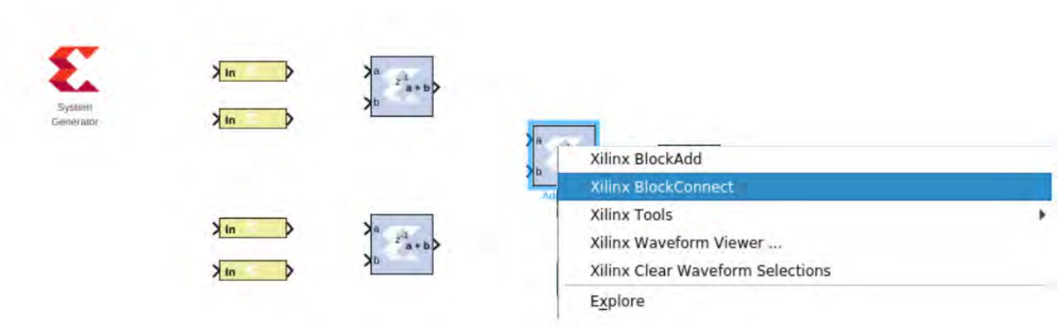
Xilinx BlockConnect

Facilitates the rapid connection of blocks in a Simulink® model.

Simple Connections

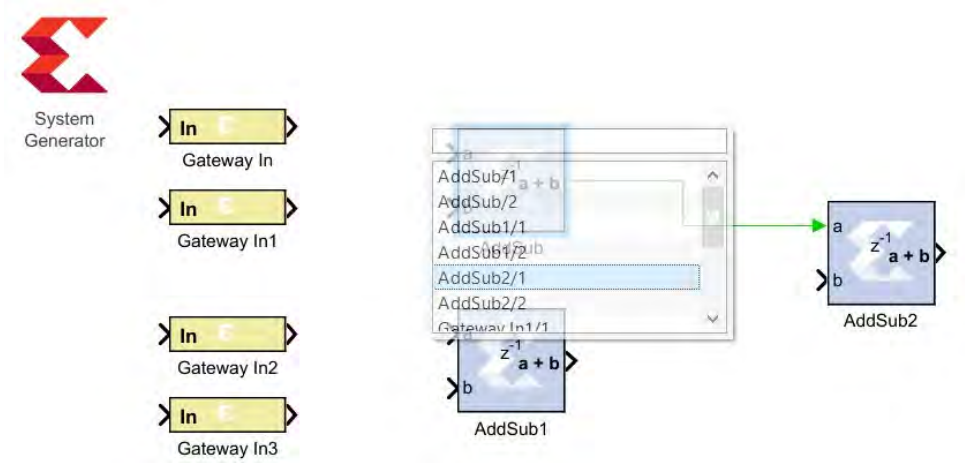
1. As shown in the following figure, select an open port of a block, right-click, keep the cursor on a particular port then right-click and select the **Xilinx BlockConnect** option to see the list of possible connections for that port. **Xilinx Block Connect** only shows the list of possible connections for one port at a time.

Figure 154: **Xilinx BlockConnect**



2. BlockConnect proposes the nearest connection with a green line. To confirm, you can double-click the selected connection in the table. The connection then turns black. Otherwise, select another connection in the table to see if the new green line connection is correct.

Figure 155: **Connecting Blocks**

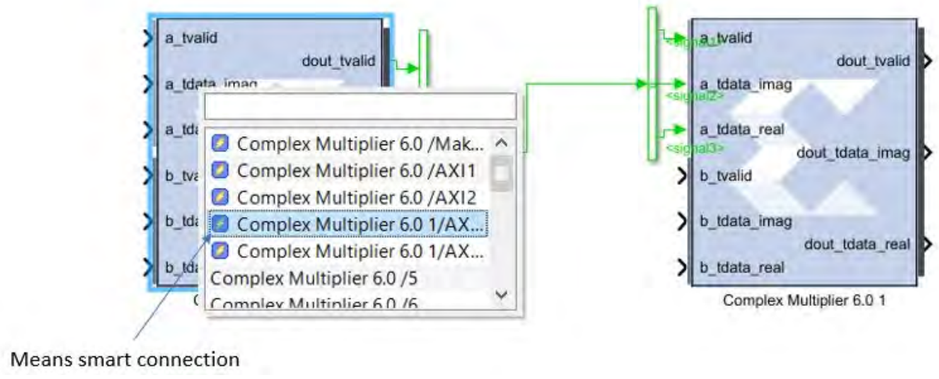


Smart Connections

As shown in the following figure, a "lightning bolt" icon indicates a "smart" connection. Smart connections have intelligence built in to help you manage the connection. For example, right-clicking a block with an AXI interface allows you to:

- Group or separate the AXI signals to or from a bus.
- Connect to other ports with the same number of AXI connections.

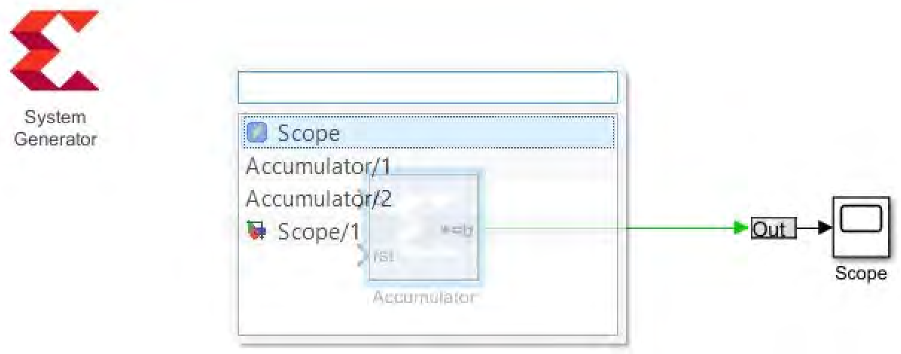
Figure 156: **Smart Connections**


a_tvalid
dout_tvalid
a_tdata_imag
Complex Multiplier 6.0 /Mak...
Complex Multiplier 6.0 /AXI1
Complex Multiplier 6.0 /AXI2
Complex Multiplier 6.0 1/AX...
Complex Multiplier 6.0 1/AX...
Complex Multiplier 6.0 /5
Complex Multiplier 6.0 /6
a_tdata_real
dout_tdata_imag
b_tvalid
b_tdata_imag
dout_tdata_real
b_tdata_real
Complex Multiplier 6.0 1
Means smart connection

No port data type checking is performed and any AXI ports with the same number of ports are allowed to connect.

In another smart connection example below, right-clicking the **Accumulator** block output, selecting **BlockConnect**, and double-clicking **Scope** creates a smart connection to the Scope block. The Gateway Out block is added automatically.

Figure 157: **Connecting Scope Block**



If a second connection is made to this Scope block, a second port is automatically added to the Scope. The driving signal name is also used to name the signal driving the scope.

Xilinx Tools > Terminate

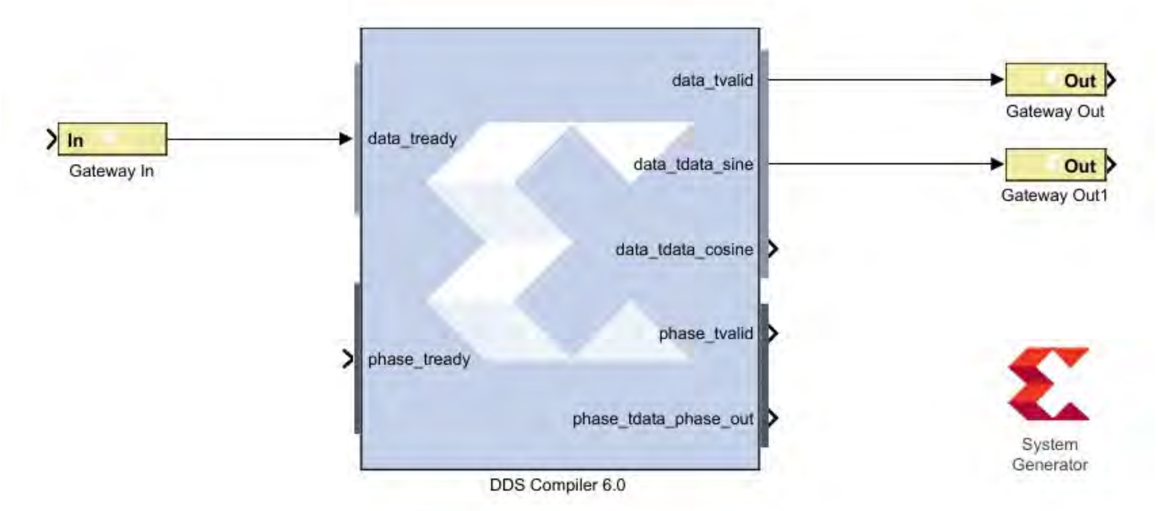
Facilitates the rapid addition of Simulink® terminator blocks on open output ports and/or Xilinx® HDL Constant Blocks on open input ports.

How to Use

Terminating Open Outputs

Consider the following model with open input and output ports:

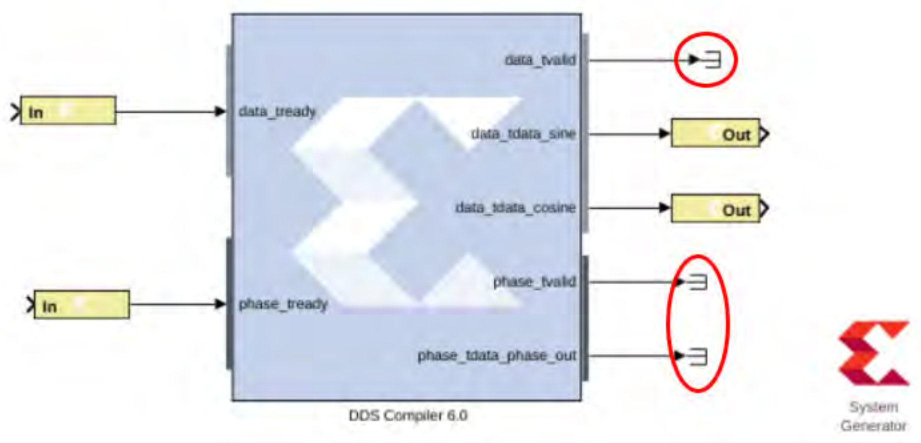
Figure 158: **Model with Open Input and Output Ports**



Right-click the DDS Compiler 6.0 block in this case and select **Xilinx Tools → Terminate → Outputs**.

The following figure illustrates the resulting terminated outputs.

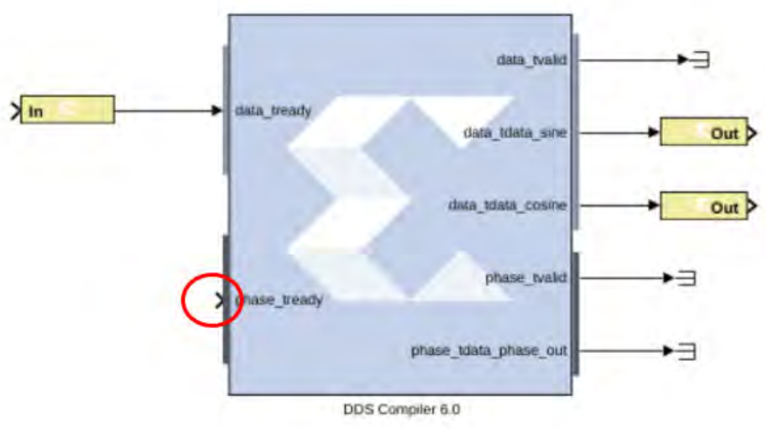
Figure 159: **Terminated Output Ports**



Terminating Open Inputs

Consider the following model with an open input port:

Figure 160: **Model with Open Input Port**

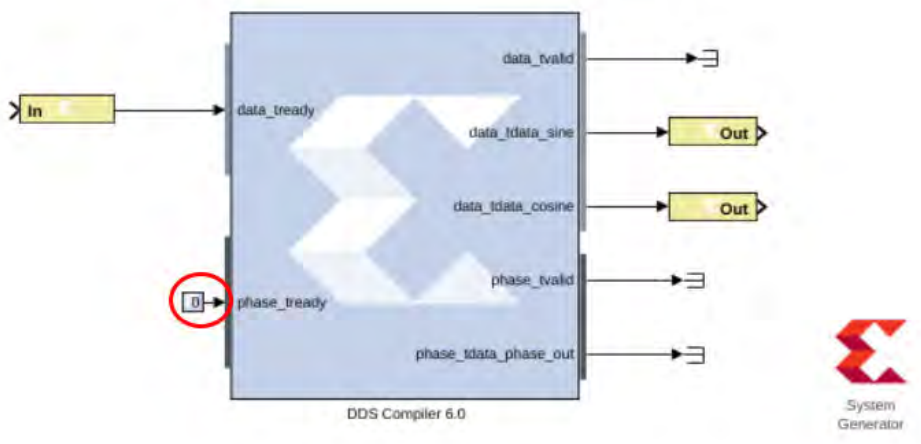


Right-click the DDS Compiler 6.0 block and select:

Xilinx Tools → Terminate → Inputs

The following figure illustrates the resulting terminated input.

Figure 161: **Terminated Input Port**



Verifying Input Port Data Type Requirements

Model Composer connects each open input port to an HDL Constant Block. The new Constant blocks are set to the following default values:

Type: Signed (2's comp)

Constant value: 0

Number of bits: 16

Binary point: 14

This terminate tool does not do data type checking on the input ports. If an open port requires a different data type, for example a Boolean data type, you will need to double-click the Constant block and change the Output Type to Boolean.

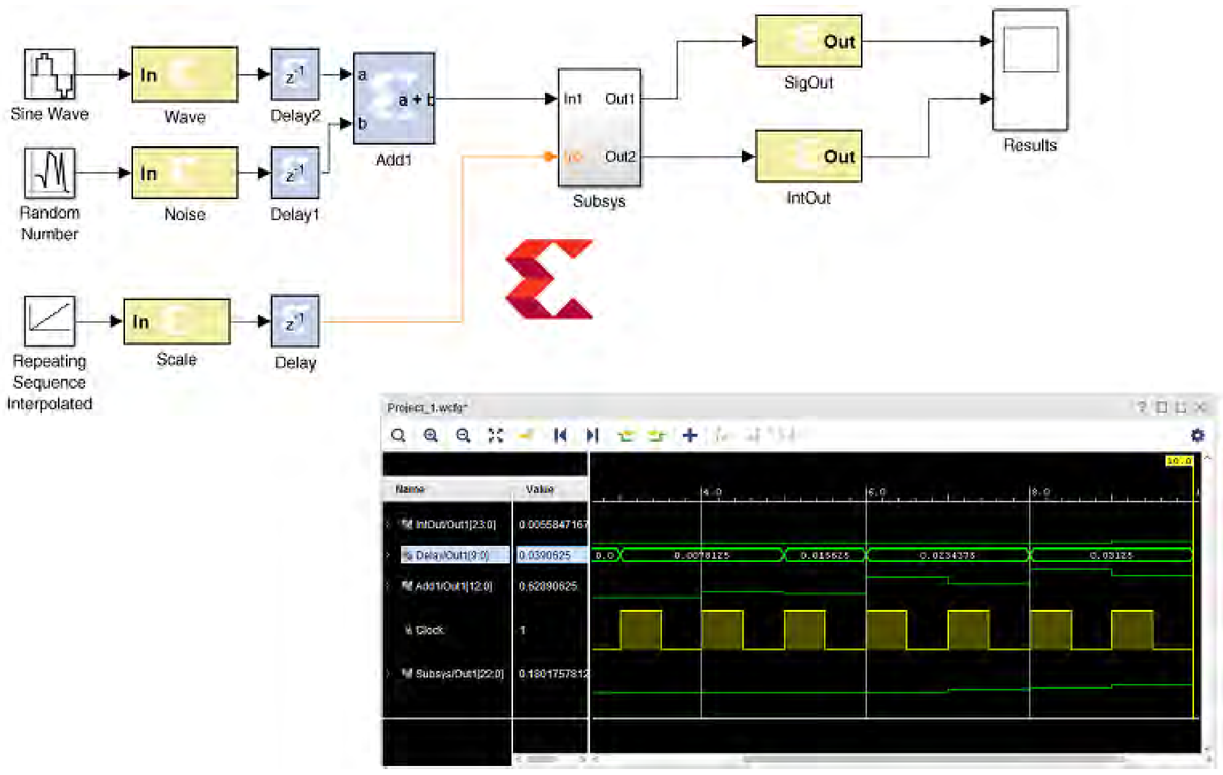
To check for data type mismatches, click the Simulink® model canvas, and enter Ctrl-D. Model Composer will report on all the data type mismatches, if there are any.

Xilinx Waveform Viewer

The Xilinx® Waveform Viewer displays a waveform diagram of selected signals in your Model Composer design. Waveforms can be displayed in the Waveform Viewer after running a Simulink simulation. Inputs and outputs of blocks in the HDL Blockset can be displayed in the Waveform Viewer.

In your design, you can select the signals that will be monitored in the Waveform Viewer. As you develop and troubleshoot your design, the waveforms for the signals you are monitoring will be updated in the Waveform Viewer each time you simulate the model.

Figure 162: **Example Design in Waveform Viewer**



The Xilinx Waveform Viewer used with Model Composer is also used by other tools in the Vivado® toolset. The Waveform Viewer is used to analyze a design and debug code in the Vivado simulator and to display data captured by the Integrated Logic Analyzer (ILA) for in-system debugging.

For information on using the Waveform Viewer to develop and troubleshoot your design, see this link in the *Vivado Design Suite User Guide: Logic Simulation* (UG900).

Waveform Viewer Files

The first time you open the Waveform Viewer for your Simulink model, Model Composer creates a `wavedata` directory in the directory containing your Simulink model.

Note: You will need write permission for the directory containing your Simulink model.

Data describing the display in the Waveform Viewer is stored in the following files in the `wavedata` directory:

- `<design_name>.wcfg` - This is the waveform configuration file. It contains the names of the signals you are monitoring in your design and how the waveforms for these signals will appear in the Waveform Viewer.
- `<design_name>.wdb` - This is the waveform database file. It contains the data necessary to draw the waveforms in the Waveform Viewer.

The names of the signals that are being monitored are stored in the Simulink model (SLX file). If the Simulink model cannot access the data in the `wavedata` directory (for example, if you moved the model's SLX file to a different directory and opened it in the new directory), you can display the monitored signals by opening the Waveform Viewer and simulating the design. The waveforms for the monitored signals will then appear in the Waveform Viewer.

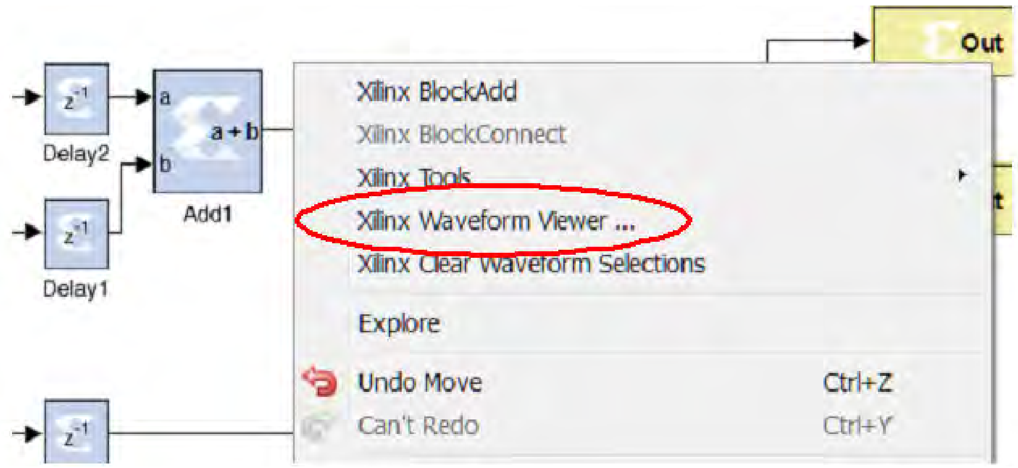
Opening the Xilinx Waveform Viewer

Youc can open the Waveform Viewer in either of the following ways:

- Opening from right-click menu:

  Right-click in your model and select **Xilinx Waveform Viewer**.

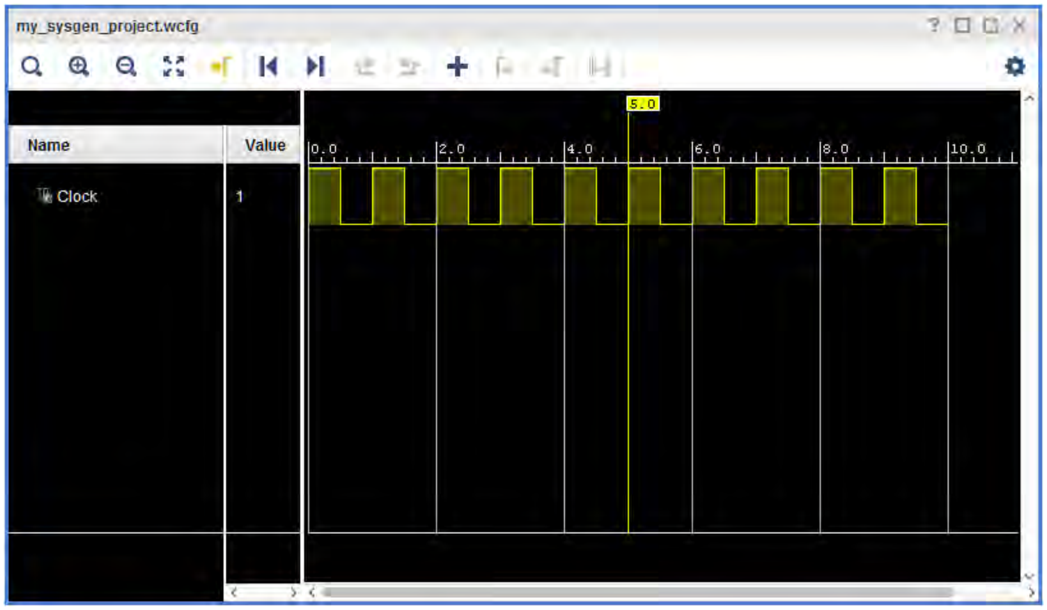
Figure 163: **Xilinx Waveform Viewer**



  If you open it from the right-click menu, the Waveform Viewer opens with the following display:

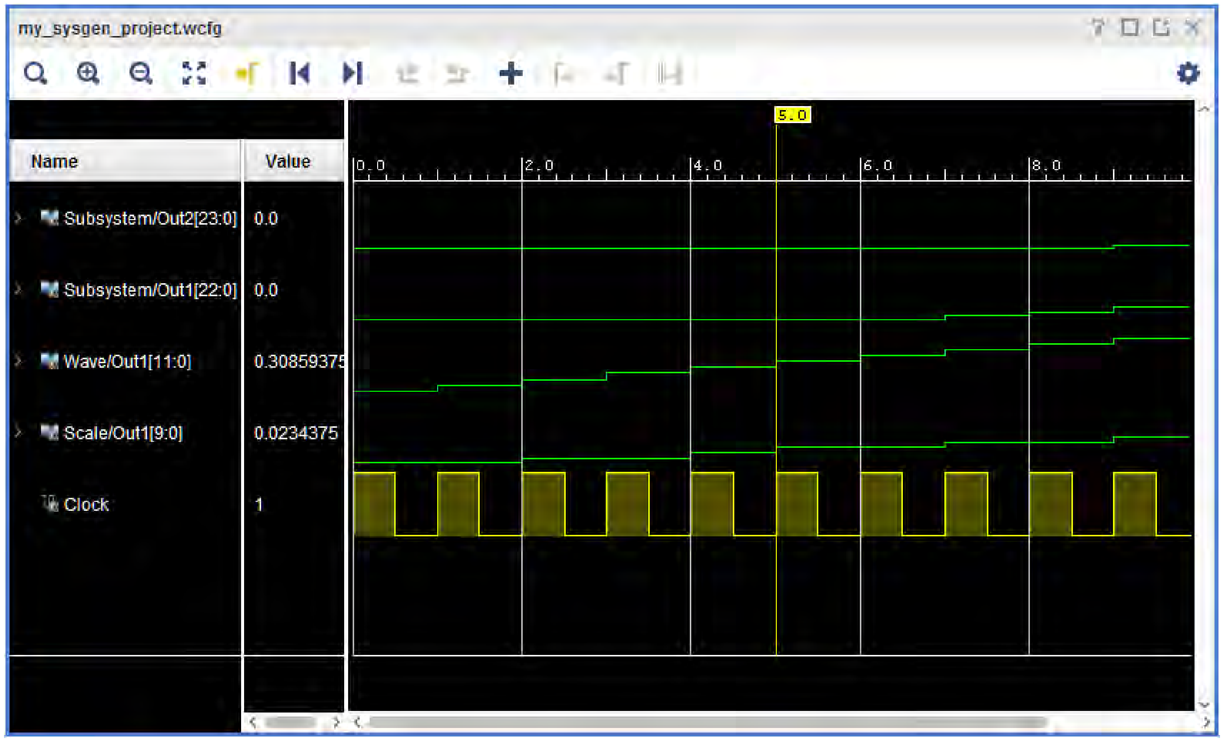
  - If this is the first time you are opening the Waveform Viewer for this design, the Waveform Viewer opens displaying waveforms for the clock signals in your design, and no other waveforms. You can then add the signals in your design that you want monitored to the Waveform Viewer display (see Adding Signals to the Waveform Viewer Display).

Figure 164: **Waveform Viewer Display**



- If you have previously monitored signals in the Waveform Viewer for this design, and have saved the data, the Waveform Viewer opens displaying the signal names, and waveforms displayed when you last closed the Waveform Viewer.

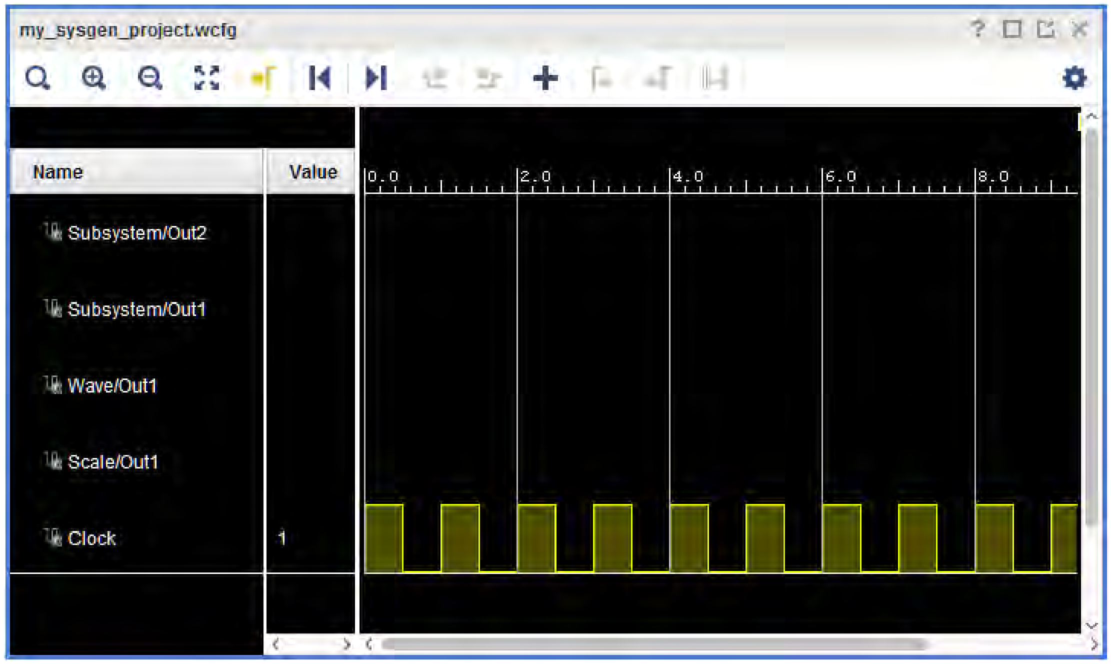
Figure 165: **Signals in Waveform Viewer**



Name	Value
Subsystem/Out2[23:0]	0.0
Subsystem/Out1[22:0]	0.0
Wave/Out1[11:0]	0.30859375
Scale/Out1[9:0]	0.0234375
Clock	1

- If you have previously monitored signals in the Waveform Viewer for this design, but cannot access the saved data (for example, if you moved the model's SLX file to a different directory, and opened it in the new directory), the Waveform Viewer opens displaying the signal names for the signals monitored when you last saved the model. The Waveform Viewer will not show the waveforms for the monitored signals until you resimulate the model.

Figure 166: **Waveform Viewer New Signals**



- Opening after simulation:

  If you have previously monitored signals in the Waveform Viewer for your design, the Waveform Viewer will open automatically when you simulate your model.

Adding Signals to the Waveform Viewer Display

Inputs and outputs of blocks in the HDL Blockset can be displayed in the Waveform Viewer. The data necessary to draw each signal's waveform is not stored with the design; it is generated by simulation. You can only display a signal's waveform after you have added the signal to the Waveform Viewer and then simulated the model.

To add signals to the display in the Waveform Viewer:

1. With the Waveform Viewer open, select a signal in the Model Composer model.

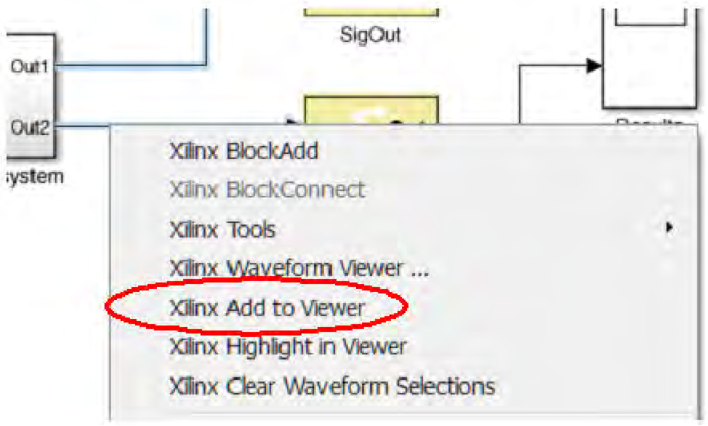
   You can also select multiple signals by using **Shift+** click to select additional signals.

   Note: For the Gateway In block, only the output signal can be displayed in the Waveform Viewer.

2. Right-click one of the selected signals in the Model Composer model and select **Xilinx Add to Viewer** in the right-click menu.

   Note: If you select a signal that is currently displayed in the Waveform Viewer, the **Xilinx Add to Viewer** entry will not appear in the right-click menu.

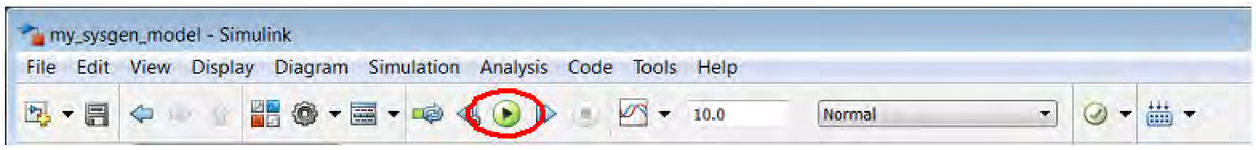
Figure 167: **Xilinx Add to Viewer**



   The signal names of the selected signals appear in the Waveform Viewer.

3. Only the names of the added signals appear in the Waveform Viewer, because the Waveform Viewer does not have the data to draw the signal's waveform until you simulate the design.
4. Simulate the model.

Figure 168: **Run Button**


my_sysgen_model - Simulink
File Edit View Display Diagram Simulation Analysis Code Tools Help
10.0
Normal

   After the simulation is finished, the waveforms for the added signals are displayed in the Waveform Viewer.

Deleting Signals From the Waveform Viewer Display

1. In the Waveform Viewer, select the signals to be deleted.

   Use **Shift+**click or **Ctrl+**click to select multiple signal names (**Ctrl+A** to select all).
2. Right click one of the selected names and select **Delete** in the right-click menu.

   OR

   Press the **Delete** key.

The waveforms are deleted from the Waveform Viewer. Deleted waveforms are no longer monitored; if you resimulate the model the deleted waveforms will not appear in the Waveform Viewer.

Cross Probing Between the Waveform Viewer and the Model

Cross probing helps you correlate the waveforms in the viewer to the wires in the Model Composer model.

You can cross probe signals between the Waveform Viewer, and the model in the following ways:

- To cross probe a signal from the Waveform Viewer to the Model Composer model, select one or more signal names in the Waveform Viewer. Use **Shift+**click or **Ctrl+**click to select multiple signal names (**Ctrl+A** to select all).

  The selected signals are highlighted in orange in the Model Composer model.

  To unhighlight a signal you have highlighted in the Model Composer model, **Ctrl+**click the signal name in the Waveform Viewer. The signal is unhighlighted in the Model Composer model.

- To cross probe a signal from the Model Composer model to the Waveform Viewer:

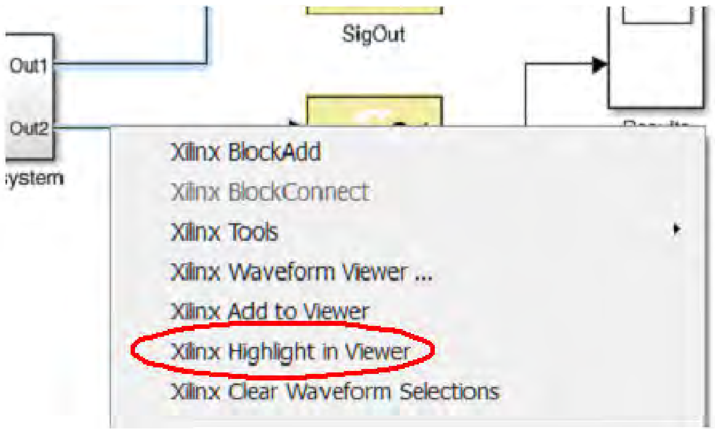
  1. With the Waveform Viewer open, select a signal in the Model Composer model.

     You can also select multiple signals by using **Shift+**click to select additional signals.

  2. Right-click one of the selected signals in the Model Composer model and select **Xilinx Highlight in Viewer** in the right-click menu.

     ***Note*:** If you select a signal that is not currently displayed in the Waveform Viewer, the **Xilinx Highlight in Viewer** entry will not appear in the right-click menu.

     Figure 169: **Xilinx Highlight in Viewer**

  3. Observe that the signal names of the selected signals are highlighted in the Waveform Viewer.

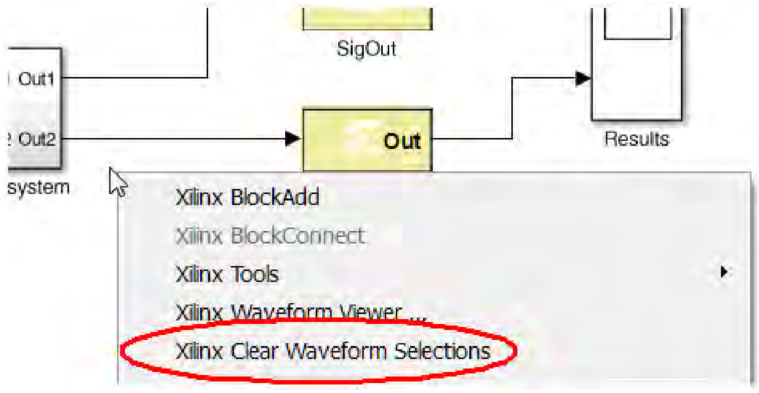
Clearing the Waveform Viewer Display

To clear the waveform display, deleting all the waveforms currently displayed in the Waveform Viewer:

1. Right-click in the Model Composer model.
2. Select **Xilinx Clear Waveform Selections** in the right-click menu.

All of the signals currently displayed in the Waveform Viewer are deleted from the Waveform Viewer display, and the Waveform Viewer closes. The deleted waveforms are no longer monitored and the `wavedata` directory (which contains data describing the current display in the Waveform Viewer) is removed from the directory containing your Simulink model.

Figure 170: **Xilinx Clear Waveform Selections**



To open the Waveform Viewer again, right-click in your model and select **Xilinx Waveform Viewer** in the right-click menu. The Waveform Viewer opens displaying waveforms for the clock signals in your design, and no other waveforms.

Customizing the Display and Analyzing Waveforms

The Waveform Viewer has many tools to customize how your waveforms are displayed and to analyze the waveforms. For information on using the Waveform Viewer to develop and troubleshoot your design, see this link in the *Vivado Design Suite User Guide: Logic Simulation* (UG900).

Tips for Working in the Waveform Viewer

The following tips will help you with your waveform analysis using the Model Composer model and the Waveform Viewer:

- Keep the Waveform Viewer open during a Model Composer session. Do not close the Waveform Viewer between each simulation.

- If you select a group of signals in the Waveform Viewer, all of the signals in the group will be cross-probed from the Waveform Viewer to the Model Composer model.
- To add multiple signals in your Model Composer model to the Waveform Viewer display, you can press and hold the left mouse button and drag the mouse to draw a box around the signals, selecting them. Then right-click one of the selected signals and select **Xilinx Add to Viewer** in the right-click menu. The selected signals will be added to the Waveform Viewer display.
- When naming an output signal for a block in your Model Composer model, avoid using the reserved characters shown in the table below. These are reserved characters in VHDL or Verilog. If your model does contain a signal with a reserved character, its name will be changed in the Waveform Viewer display according to the following mapping table.

Table 7: **Reserved Characters**

Reserved Character	Mapped To
(	#1
)	#2
[	#3
]	#4
.	#5
,	#6
:	#7
\	#8

Closing the Waveform Viewer

To close the Waveform Viewer, select **File→Exit**. If you have not yet saved the waveform data, you will be prompted to save the data before the Waveform Viewer closes.

HLS Library

Introduction

Vitis™ Model Composer provides the HLS blockset in the Xilinx toolbox. This enables you to transform your algorithmic specifications to production-quality IP implementations using automatic optimizations and leveraging the high-level synthesis technology of Vitis HLS. Using the IP integrator in Vivado, you can then integrate the IP into a platform that, for example, may include a Zynq® device, DDR3 DRAM, and a software stack running on an Arm® processor.

The HLS library in the Xilinx Tool Box provides optimized blocks for use within the Simulink environment. These include basic functional blocks for expressing algorithms like Math, Linear Algebra, Logic, and Bit-wise operations and others.

The HLS library contains the following categories of elements.

Table 8: **HLS Block Library**

Library	Description
Logic and Bit Operations	Blocks that supports the compound logical operations and bit-wise operations.
Lookup Tables	Block set that performs a one dimensional lookup operation with an input index.
Math Functions	Blocks that implement mathematical functions.
Ports and Subsystems	Blocks that allow creation of subsystems and input/output ports.
Relational Operations	Block set to define some kind of relation between two entities (e.g., Numerical Equality and inequalities).
Signal Attributes	Includes block which helps to maintain the compatibility between input type and output type (e.g., Type casting).
Signal Operations	Blocks that support simple modifications to the time variable of the signal to generate new signals (e.g., Unit Delay).
Signal Routing	Blocks that supports the setup to track signal sources and destinations (e.g., Bus selector).
Sinks	Include blocks that receive physical signal output from other blocks.
Source	Include blocks that generate or import signal data.
Tools	Include blocks that controls the implementation/Interface of the Model.

For information on specific blocks in the Model Composer HLS library, see HLS Blockset.

The HLS block library is compatible with the standard Simulink block library, and these blocks can be used together to create models that can be simulated in Simulink. However, only certain Simulink blocks are supported for code generation by Model Composer. The Simulink blocks compatible with output generation from Model Composer can be found in the HLS block library.

Model Composer also lets you create your own custom blocks from existing C/C++ code for use in models. Refer to Importing C/C++ Code as Custom Blocks for more information.

Your Model Composer design is bit-accurate with regard to the final implementation in hardware, though untimed. You can compile the design model into C++ code for synthesis in Vitis HLS, create HDL blocks, or create packaged IP to be used in Vivado.

To familiarize yourself with the Model Composer HLS library, the *Vitis Model Composer Tutorial* (UG1498) includes labs and data to walk you through the tool.

The rest of this document discusses the following topics:

- Creating a Model Composer model using HLS block libraries.
- Importing existing C-code into the Model Composer HLS block library for use in your models.
- Compiling the model for use in downstream design tools.
- Verifying your Model Composer model, C++, and RTL outputs.

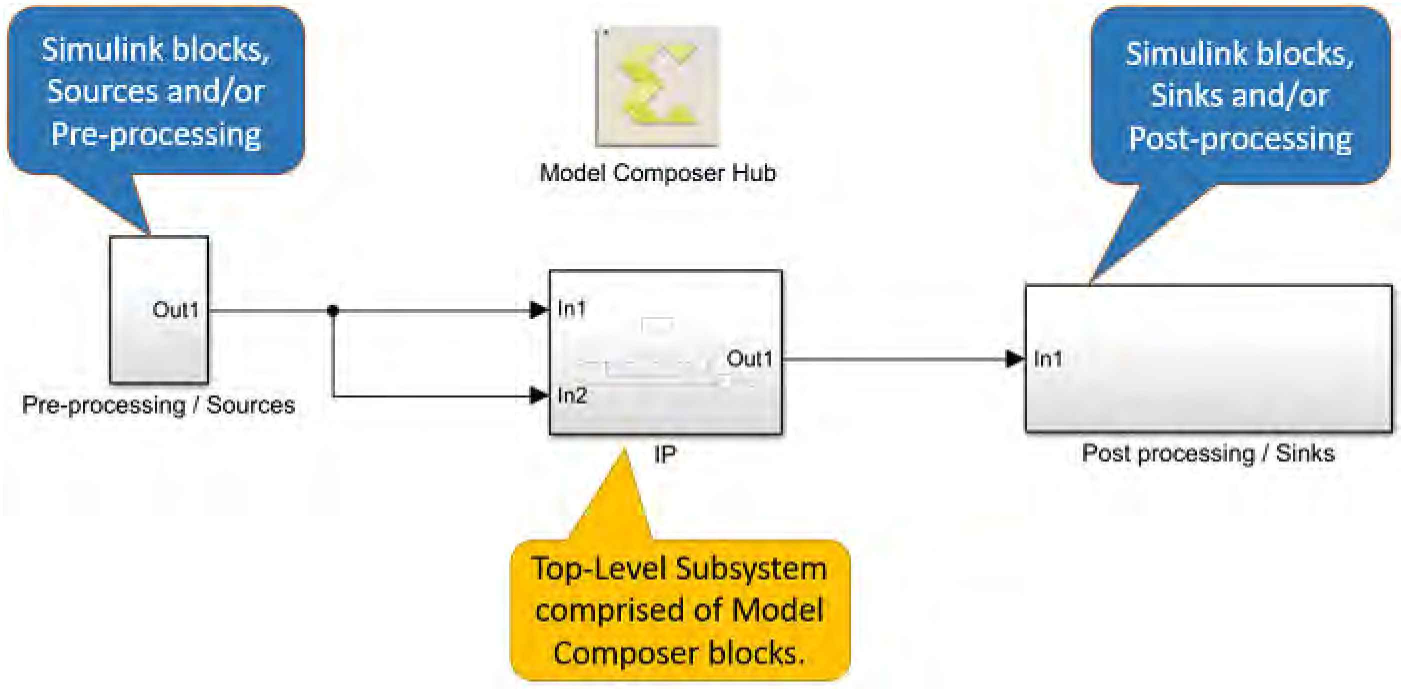
Creating a Model Composer Design

As shown in the image below, a Model Composer design with HLS blocks includes the following elements:

1. Simulink® blocks that determine inputs, and provide source signals. These blocks are used in simulation of the design, but do not affect the output generated by Model Composer.
2. A top-level subsystem block, as described in Creating a Top-Level Subsystem Module, that encapsulates the algorithm defined by the Model Composer model. This subsystem module can contain:
   - Blocks from the HLS library to define your algorithm, as listed in the HLS Library.
   - Custom imported functions as described in Importing C/C++ Code as Custom Blocks.
   - An Interface Spec block that defines the hardware interfaces as described in Defining the Interface Specification.
3. The Model Composer Hub block that controls throughput of the design, and output generation through a series of options as described in Adding the Model Composer Hub.

4. The output signals, or sinks that process the output in Simulink. Again, these blocks are used during Simulink simulation as described in Simulating and Verifying Your Design, but do not affect the output generated by Model Composer.

Figure 171: **Elements of a Model Composer Design**


Simulink blocks, Sources and/or Pre-processing
Model Composer Hub
Simulink blocks, Sinks and/or Post-processing
Out1
Pre-processing / Sources
In1
In2
Out1
IP
In1
Post processing / Sinks
Top-Level Subsystem comprised of Model Composer blocks.

Creating a New Model

You create a new model by adding blocks from the Library Browser into theSimulink Editor. You then connect these blocks with signal lines to establish relationships between blocks. The Simulink Editor manages the connections with smart guides and smart signal routing to control the appearance of your model as you build it. You can add hierarchy to the model by encapsulating a group of blocks and signals as a subsystem within a single block. Model Composer provides a set of predefined blocks that you can combine to create a detailed model of your application.

In the Simulink start page, select **Blank Model** to open a new model.

TIP: *You can also open an existing Model Composer template if any have been defined. Model templates are starting points to reuse settings and block configurations. To learn more about templates, see Create a Template from a Model in the Simulink documentation.*

The Simulink start page also lists the recent models that you have opened on the left-hand column. You can open one of these recent models if you prefer.

The blank model opens, and you will create the Model Composer model by adding blocks, specifying block parameters, and using signal lines to connect the blocks to each other.

IMPORTANT! *HLS Library only supports one sample time for the model, and does not support multi-time systems. All HLS Library blocks inherit the sample time from the source block of the model. See What is Sample Time for more information.*

To save the model select **File → Save** from the main menu. The **Save As** dialog box is opened, with a file browser. Navigate to the appropriate folder or location, and enter a name for the model in the **File Name** field. Click **Save**. The model is saved with the file extension `.slx`.

Model Composer also includes example models based on HLS library which can be accessed from the *Model Composer Examples* section of the *Vitis Model Composer* documentation available from the **Help** menu in the tool, or by typing the `xmcOpenExample` command from the MATLAB command prompt:

```
>> xmcOpenExample
```

This command opens a window to specify the target directory for downloading the examples from the GitHub repository.

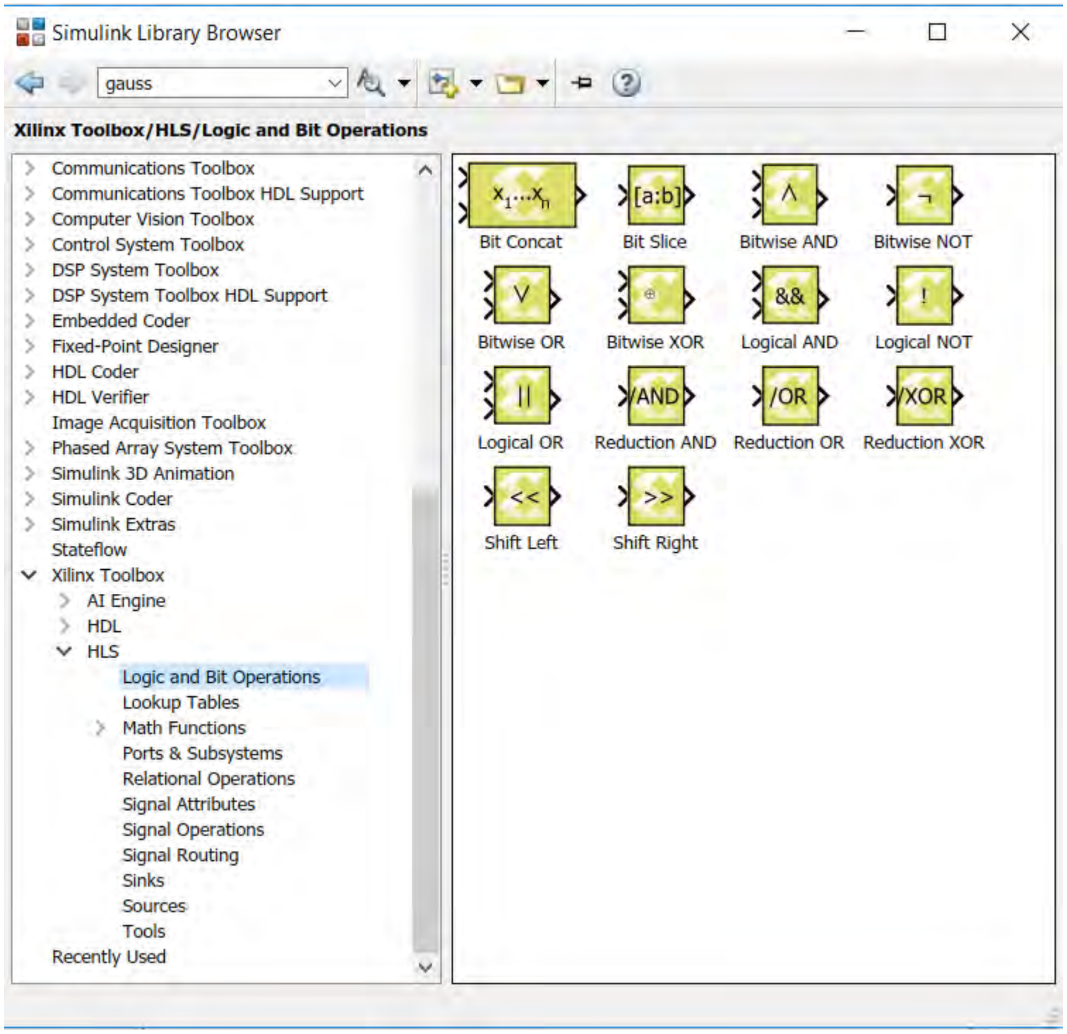
Adding Blocks to a Model

You can add blocks to the current model by opening the Library Browser and dragging and dropping the block onto the design canvas of the Simulink Editor. Open the Library Browser by clicking the button, or by selecting the **View → Library Browser** command from the main menu. You will see the standard Simulink library of blocks, as well as the HLS library in the Xilinx Toolbox.

TIP: *You can also open the Library browser by typing the slLibraryBrowser command from the command prompt.*

The HLS blocks are organized into sub-categories based on functionality. The figure below shows the **HLS → Logic and Bit Operations** block library in the Library Browser.

Figure 172: **Library Browser**



Double-clicking a block in the Library Browser, opens the Block Parameter dialog box displaying the default values for the various parameters defined on the selected block. While the block is in the library, you can only view the parameters. To edit the parameters, you must add the block to the design canvas.

To get additional information about a block you can right-click a block in the Library Browser and select the **Help** command. Alternatively, you can double-click the block in the Library Browser and click the **Help** button from the block dialog box. The Help browser opens with specific information for the block.

When you drag and drop the block onto the canvas, the block is added to the model with the default parameter values defined.

TIP: *You can also quickly add blocks to the current model by single-clicking on the design canvas of the Simulink Editor and typing the name of a block. Simulink displays possible matches from the libraries, and you can select and add the block of interest.*

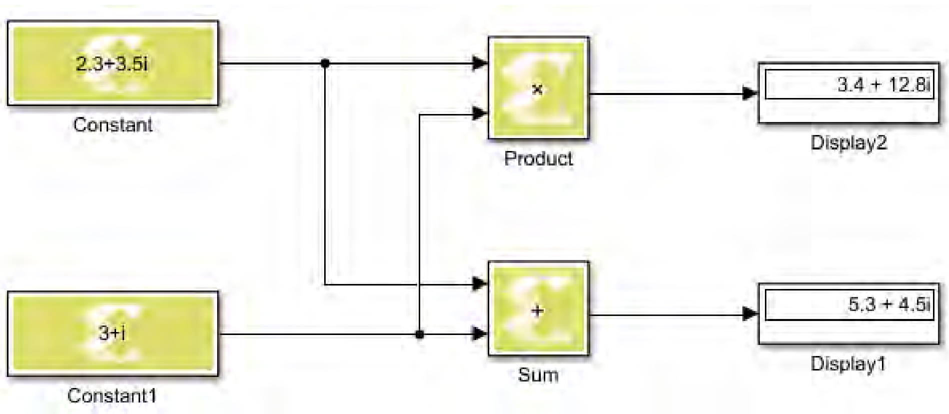
Simulink models contain both signals and parameters. Signals are represented by the lines connecting blocks. Parameters are coefficients that define key characteristics and behavior of a block.

Connecting Blocks

You can connect the output ports on blocks to the input ports of other blocks with signal lines. Signal lines define the flow of data through the model. Signals can have several attributes:

- Data type: Defines the type of data carried by the signal. Values can range from integer, to floating point, to fixed point data types. See Working with Data Types for more information.
- Signal dimension: Defines the values as being scalar, vector, or matrices. See Signal Dimensions and Matrices, Vectors, and Scalars for more information.
- Complexity: Defines a value as being a complex or real number. See Signal Values for more information. The figure below shows complex numbers propagating through a model.

Figure 173: **Complex Signal Values**



To add a signal line, position the cursor over an input or output port of a Simulink block. The cursor changes to a cross hair (+). Left-click and drag the mouse away from the port. While holding down the mouse button, the connecting line appears as a dotted line as you move across the design canvas. The dotted line represents a signal that is not completely connected.

Release the mouse button when the cursor is over a second port to be connected. If you start with an input port, you can stop at an output port, or connect to another signal line; if you start at an output you can stop at an input. Simulink connects the ports with a signal line and an arrow indicating the direction of signal flow.

You can connect into an existing line by right-clicking and dragging the mouse. This creates a branching line connected to the existing signal line at the specified location. The branch line can connect to an input or output as appropriate to the connected signal.

> **TIP:** *You can also connect blocks by selecting them sequentially while holding the Ctrl key. This connects the output of the first block into the input of the second block. Keeping the Ctrl key pressed and selecting another block continues the connection chain.*

Simulink updates the model when you **Run** simulation. You can also update the model using the **Simulation → Update Diagram** menu command, or by typing `Ctrl+D` at any point in the design process. You will see the Data Types, Signal Dimensions and Sample Times from the source blocks propagate through the model.

> **TIP:** *You can use the* ***Display → Signals and Ports*** *menu command to enable the various data that you want displayed in your model, such as* ***Signal Dimensions*** *and* ***Port Data Types****.*

You cannot specify sample times on HLS Library blocks, except for the Constant block from the Source library of the HLS library. Model Composer infers the sample time from the source blocks connected at the input of the model, and does not support multiple sample times in the Model Composer design.

By updating the diagram from time to time, you can see and fix potential design issues as you develop the model. This approach can make it easier to identify the sources of problems by limiting the scope to recent updates to the design. The **Update Diagram** is also faster than running simulation.

Working with Data Types

Data types supported by HLS library blocks include the following:

Table 9: **Model Composer Data Types**

Name	Description
double	Double-precision floating point
single	Single-precision floating point
half*	Half-precision floating point
int8	Signed 8-bit integer
uint8	Unsigned 8-bit integer
int16	Signed 16-bit integer
uint16	Unsigned 16-bit integer
int32	Signed 32-bit integer
uint32	Unsigned 32-bit integer
fixed*	Signed and unsigned fixed point
boolean	For this data type, Simulink represents real, nonzero numeric values as TRUE (1)

IMPORTANT! *Data types marked with '*' are specific to Model Composer HLS Library, and are not naturally supported by Simulink. While Simulink does support fixed point data types, you must have the Fixed-Point Designer™ product installed and licensed. In addition, the fixed point data type supported by Vitis Model Composer is not compatible with the fixed point data type supported by Simulink although it uses a similar notation.*

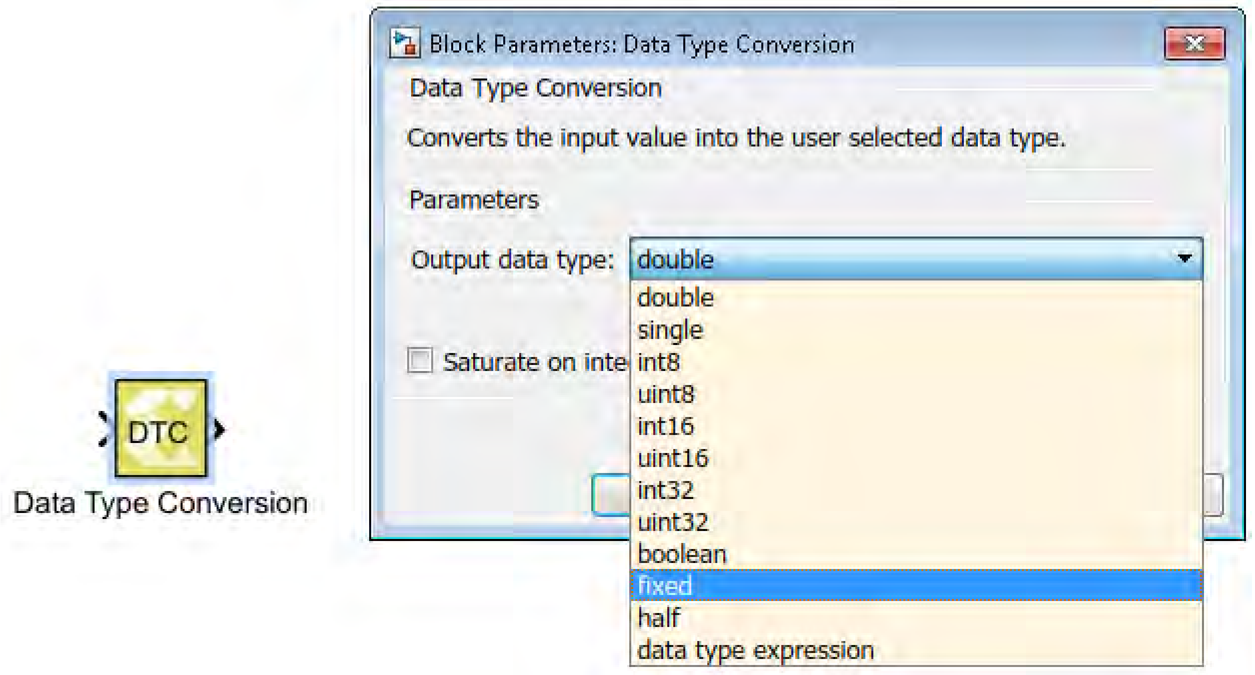
Notice in the preceding table there are some data types that are supported by Model Composer HLS Library that are not supported by default in Simulink. If you connect blocks from the HLS library, with `fixed` or `half` data types, to Simulink native blocks, you will see an error when running simulation in Simulink, or when using the **Update Diagram** command, or pressing `Ctrl+D`.

```
RelationalOperator does not accept signals of data type 'x_sfix16'.
'ConstRE_or_IMpartBug/Relational Operator' only accepts numeric and
enumerated data types.
```

This error indicates that Simulink could not cast the signal value from the Model Composer fixed data type to a double precision floating point data type.

In cases of mismatched data types, Model Composer recommends that you use a Data Type Conversion block to specify the behavior of the model, and indicate the conversion of one data type to another. The Data Type Conversion block (DTC) is found in the HLS Library under the Signal Attributes library.

Figure 174: **Data Type Conversion Block**



Block Parameters: Data Type Conversion

Data Type Conversion

Converts the input value into the user selected data type.

Parameters

Output data type: double

- double
- single
- int8
- uint8
- int16
- uint16
- int32
- uint32
- boolean
- fixed
- half
- data type expression

Saturate on inte

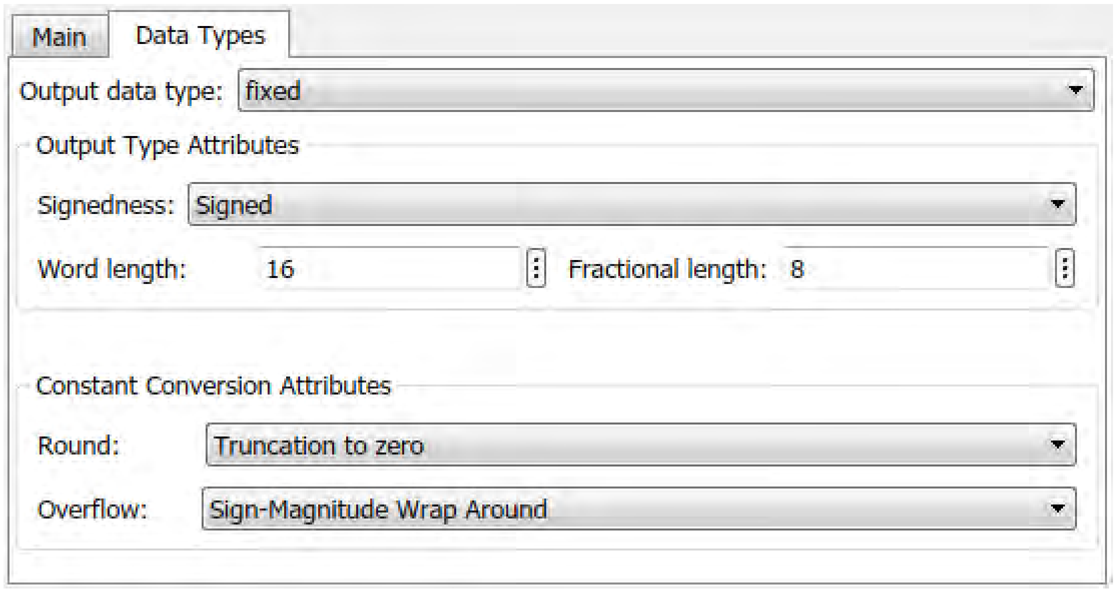
The DTC block lets you specify the **Output data type**, while the input data type is automatically determined by the signal connected to the input port. Using the DTC block, you can convert from single precision floating point to double precision for example, or from double precision to single precision.

IMPORTANT! *You must exercise caution when casting from a higher precision data type to a lower precision data type, as loss of precision can lead to rounding or truncation and loss of data can occur.*

Working with Fixed-Point Data Types

As indicated earlier, Simulink® provides support for fixed-point data types through the Fixed-Point Designer™ product. However, the format of the `fixed` data type supported by Model Composer and Simulink are not compatible.

Figure 175: **Fixed-Point Data Type**



Output data type: fixed
Output Type Attributes
Signedness: Signed
Word length: 16
Fractional length: 8
Constant Conversion Attributes
Round: Truncation to zero
Overflow: Sign-Magnitude Wrap Around

The format used to display the Model Composer fixed-point data types is as follows: `x_[u/s]fix[wl]_E[n][fl]`

Where:

- `x_`: Is the prefix indicating the Xilinx fixed data type.
- `[u/s]`: Represents signed or unsigned data.
- `fix`: Indicates the fixed-point data type.
- `[wl]` Specifies the word length of the data.
- `E`: Prefix for the fractional portion of the fixed-point data type. Does not display if the fractional length is 0.

- `n`: Displays 'n' if the binary point is to the left of the right-most bit in the word; or displays no 'n' if the binary point is to the right of the right-most bit in the word.
- `[fl]`: Specifies the fractional length of the fixed-point data type, indicating the position of the binary point with respect to the right-most bit in the word.

For example, `x_sfix16_En6` represents a signed 16-bit fixed-point number, with 6-bits allocated to the right of the binary point.

Notice the fixed-point data type also lets you specify what happens in the case of data overflow, or the need to do rounding or truncation. For more information refer to Data Type Conversion.

You must use a DTC block to convert the Model Composer fixed-point data type into the Simulink fixed-point data type. However, there is no direct conversion between Model Composer fixed-point and Simulink fixed-point data types, so you can use the following method:

1. Convert Model Composer fixed-point data type to the `double` data type using the DTC block from the HLS library in the Library Browser.
2. Convert the `double` data type to Simulink format fixed-point data type using the Simulink Data Type Conversion block from the Simulink Signal Attributes library in the Library Browser.
3. Match the signedness, word length, and fractional length between the two fixed-point data types.

TIP: *Converting between the Model Composer fixed data type and the Simulink fixed data type is not recommended unless necessary for your design. You can convert from the Model Composer fixed-point data type to double, as shown in the first step, and this should be sufficient for most applications.*

Although handling fixed-point data type in a design is more time consuming, the value of using fixed-point data types for applications targeted at implementation in an FPGA is worth the challenge. It is widely accepted that designing in floating point leads to higher power usage for the design. This is true for FPGAs where floating-point DSP blocks have been hardened onto the FPGA and users must implement a floating point solution using DSP blocks and additional device resources. Floating-point implementations require more FPGA resources than an equivalent fixed-point solution. With this higher resource usage comes higher power consumption and ultimately increased overall cost of implementing the design. For more information refer to the white paper *Reduce Power and Cost by Converting from Floating Point to Fixed Point* (WP491).

Working with Half Data Types

Model Composer also supports a `half` precision floating point data type, which is 16 bits wide instead of 32 bits, because it consumes less real estate on the target device when implemented on the FPGA. This is an important consideration when designing in Model Composer. However, Simulink does not support the `half` data type, and this can lead to errors in simulation. The solution is to use the Model Composer DTC block to convert the `half` into the `single` data type supported by Simulink, for those portions of the design that are not in the Model Composer sub-module and will not be part of the generated output.

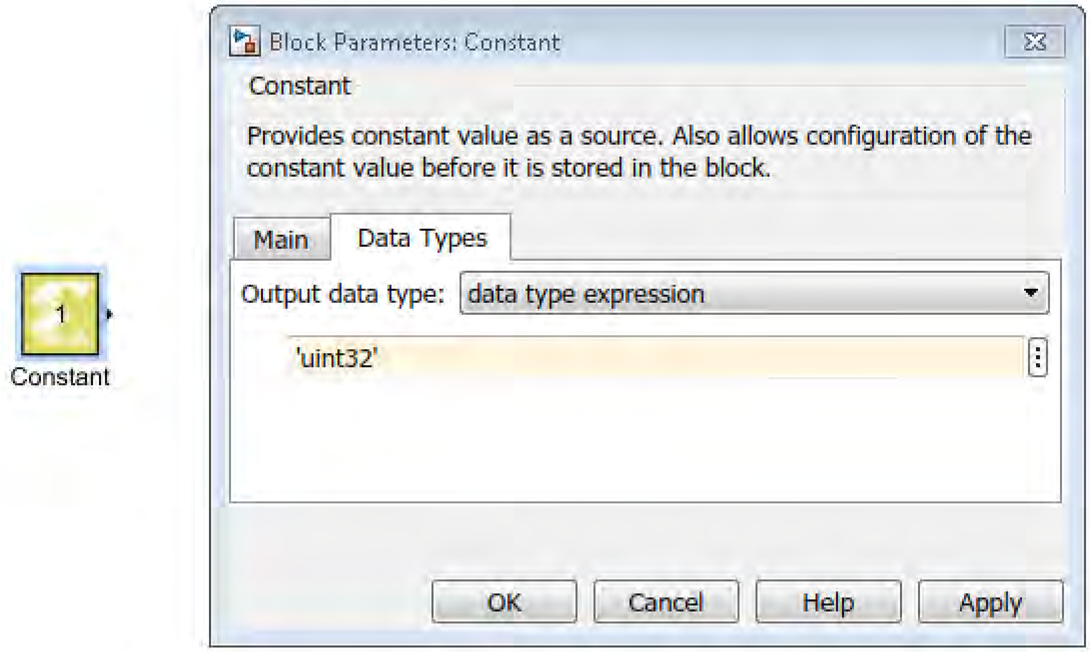
Working with Data Type Expression

Model Composer lets you specify data types as an expression. Currently the following HLS library blocks support data type expression:

- Constant
- Data Type Conversion
- Gain
- Look-Up Table
- Reinterpret

To specify a data type expression open one of the block types that support it, and edit the data type and value. The following shows a data type expression being defined for the Constant block. The **Output data type** is specified as an expression, and a string is specified to indicate the data type value, in this case 'uint32'.

Figure 176: **Data Type Expression**



Constant

Provides constant value as a source. Also allows configuration of the constant value before it is stored in the block.

Output data type: data type expression

'uint32'

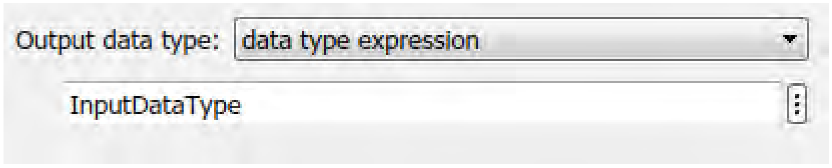
The data type value can be specified as a string representing any of the supported data types shown in the Model Composer Data Types table in the Working with Data Types section. The exception to this rule is for fixed point data types, which are not specified with the `fixed` string, but are defined according to the display format discussed under Working with Fixed-Point Data Types (e.g. 'x_sfix16_En8').

The real benefit of defining a data type as an expression is the ability to programmatically determine the data type value using a variable from the model. For instance, if you define a variable from the MATLAB® command line:

```
>> InputDataType = 'x_ufix8_En7';
```

You can use the variable in defining the data type expression.

Figure 177: **Variable Data Type**



Output data type: data type expression

InputDataType

You can specify variables from the MATLAB command line, or define variables within the model using the **Tools→Model Explorer** menu command, or simply pressing `Ctrl+H`. From the Model Explorer you can create, edit, and manage variables that the model or a block uses.

TIP: *You can also enable the* ***View → Property Inspector*** *command to display the variables for currently selected objects.*

Managing Overflow

Sometimes the data type definition for a block will not support the incoming data value on a signal. In these cases, an overflow condition can occur if the value on the signal is too large or too small to be represented by the data type of the block. The two types of overflow that can occur are wrap overflow, and saturation overflow:

- **Wrap on Overflow**: This is the default overflow mechanism, and causes the bit value to wrap at the overflow point. For instance, when an arithmetic operation such as multiplying two numbers produces a result larger than the maximum value for the data type, effectively causing a wrap around.
- **Saturate on Overflow**: This is an overflow mechanism in which all operations such as addition and multiplication are limited to a fixed range between the minimum and maximum values supported by the data type. In essence, the value will reach the maximum or minimum value and stop.

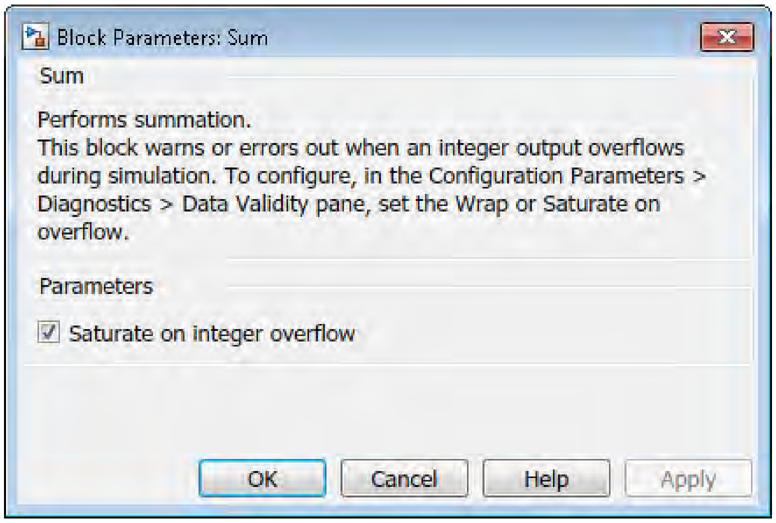
Wrapping is the default method for handling overflow, as it occurs naturally as the value overruns the data type. There is no checking required. However, the **Saturate on Overflow** option requires some additional logic in order to check the data value against the permitted maximum or minimum value to prevent wrapping. This additional logic consumes available resources on the target device.

Saturate on Overflow

Saturate on Integer Overflow

Model Composer currently supports overflow detection of integer data values on signals. As previously indicated, the default overflow mechanism is to wrap on overflow. Specific blocks in the standard Simulink library of blocks and in the HLS library have an option to **Saturate on integer overflow**. This can be enabled on the Block Parameters dialog box. This parameter applies only if the output is an integer (int8, int16, int32, uint8, uint16, uint32). Refer to the Model Composer Block Library for information specific to a block.

Figure 178: **Saturate on Integer Overflow**



Block Parameters: Sum

Sum

Performs summation.
This block warns or errors out when an integer output overflows during simulation. To configure, in the Configuration Parameters > Diagnostics > Data Validity pane, set the Wrap or Saturate on overflow.

Parameters

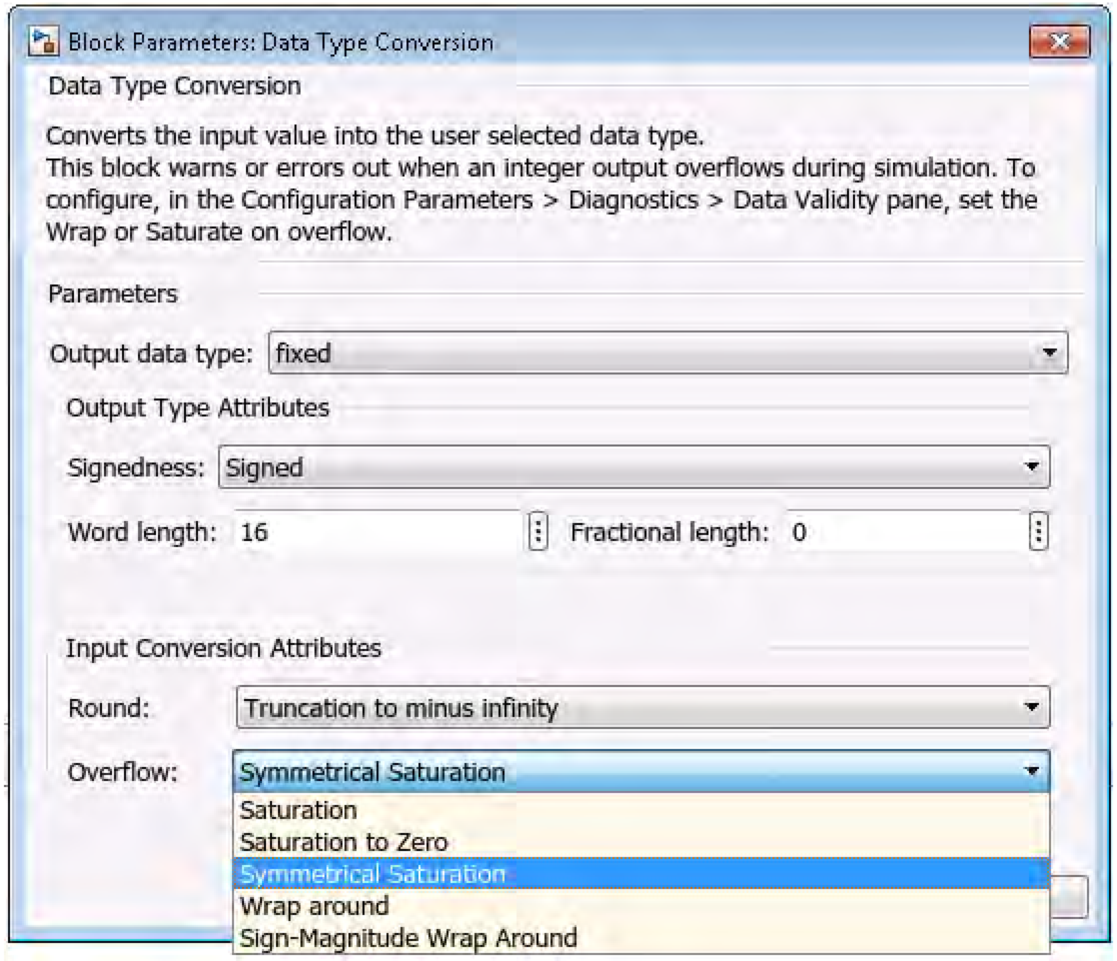
☑ Saturate on integer overflow

Saturate on integer overflow simply means that when the input value exceeds the range of values supported by the output, either too great or too small, the value simply sits at the max or min supported value. The value is saturated, and does not change.

Saturate on Fixed-Point Overflow

For fixed-point data types, as supported on the Data Conversion Block (DTC) for example, the overflow modes offer more control than the Saturate on integer overflow option, as shown in the following figure.

Figure 179: **Fixed-Point Overflow**



Block Parameters: Data Type Conversion

Data Type Conversion

Converts the input value into the user selected data type.
This block warns or errors out when an integer output overflows during simulation. To configure, in the Configuration Parameters > Diagnostics > Data Validity pane, set the Wrap or Saturate on overflow.

Parameters

Output data type: fixed

Output Type Attributes

Signedness: Signed

Word length: 16 Fractional length: 0

Input Conversion Attributes

Round: Truncation to minus infinity

Overflow: Symmetrical Saturation

A description of the different fixed-point overflow modes is provided below, with a graph to illustrate the condition.

Table 10: **Fixed-Point Overflow Modes**

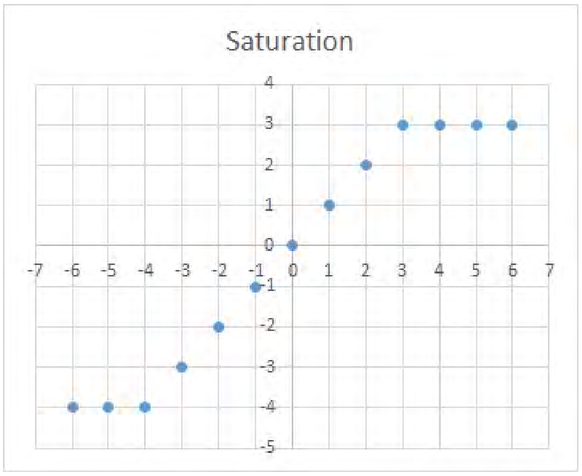
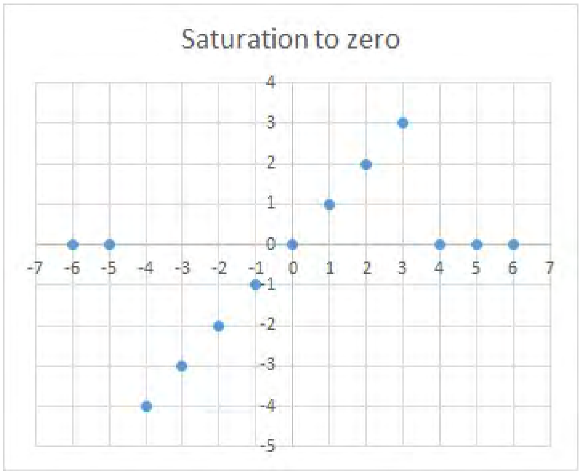
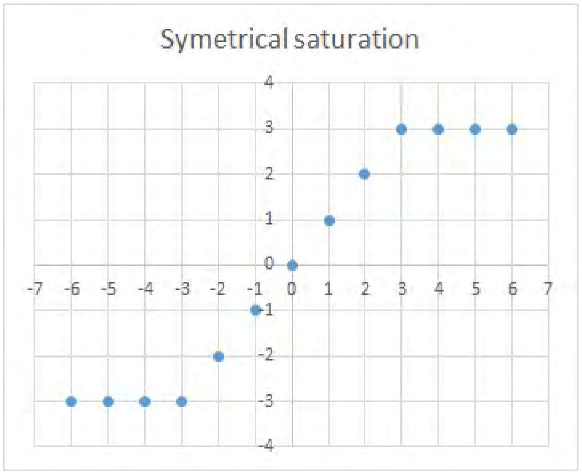
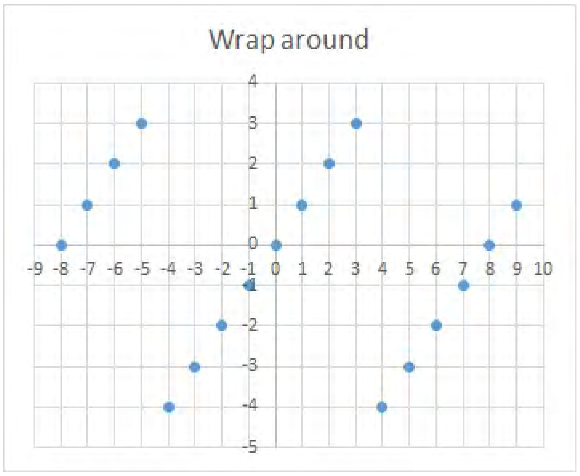
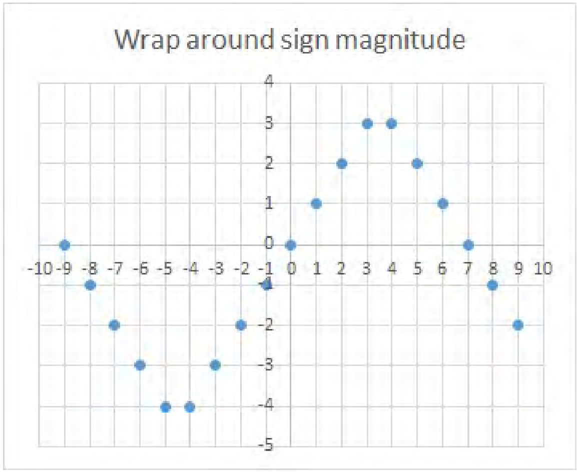
Mode	Description	Image
Saturation	When the input value overflows the output data type, the output value reaches saturation at the min or max value, and does not change.	
Saturation to Zero	When the input value overflows the output data type, the output value reaches saturation at the min or max value, and returns to zero.	
Symmetrical Saturation	Like Saturation to Zero, except the min and max values are symmetrical, or equal in size though opposite in value.	

Table 10: **Fixed-Point Overflow Modes** *(cont'd)*

Mode	Description	Image
Wrap around	When the input value exceeds the output data type, the output value is wrapped from the maximum value to the minimum value, or from the minimum value to the maximum value, thus cycling through the range of permitted values.	
Sign-Magnitude Wrap Around	When the input value exceeds the output data type, the output value reaches the maximum value, and then begins decreasing to return to the minimum value. In an underflow situation, the minimum value is reached, and begins increasing to return to the maximum value.	

Configuring Overflow Warnings

In case of either wrap or saturate, you may want to know when overflow occurs. You can define how Simulink handles each of these overflow conditions in the model by clicking on the **Model Configuration Parameters** command ( ) on the tool bar menu, or typing `Ctrl-E`. In the Configuration Parameters dialog box, under the **Diagnostics → Data Validity** tab, you can specify values for the **Wrap on Overflow** and **Saturate on Overflow** fields. Each of these fields can have one of the following settings:

- **none**: Simulink takes no special action to report or handle the overflow.
- **warning**: A message will be displayed in the diagnostic viewer. The next warning for the same block will be ignored, and simulation will continue.

- **error**: An error message will be displayed in the diagnostic viewer and the simulation will be terminated.

TIP: *Help for this dialog box can be found under Model Configuration Parameters: Data Validity Diagnostics.*

Creating a Top-Level Subsystem Module

In order to generate output from the Model Composer model the top-level of the Model Composer model must contain the Model Composer Hub block, as described in Adding the Model Composer Hub, as well as a subsystem that encapsulates the application design. To generate output from the subsystem that is instantiated at the top-level of the design, only HLS Library blocks and a limited set of specific Simulink® blocks can appear in the subsystem. The HLS Library blocks define the functions to be compiled for the packaged IP or compiled C++ code. The top-level design can contain other blocks and subsystem modules that serve different purposes, such as simulation, but the primary application must be completely contained within the specified subsystem.

TIP: *The specific Simulink blocks that are supported in the Model Composer subsystem also appear in the Model Composer block library. Refer to Supported Simulink Blocks for a complete list.*

To create a subsystem from within a model, add one or more blocks to the model canvas, select the blocks, and turn the selected blocks into a subsystem:

1. Drag and drop blocks onto the model canvas in the Simulink Editor, as explained in Adding Blocks to a Model.
2. Select one or more blocks, and right-click to use the **Create Subsystem from Selection** command.
3. Name the subsystem, giving it the same name you want to assign to the generated output application or IP.
4. Double-click the subsystem to open it in the Simulink Editor, and continue the design.

The Explorer bar and Model Browser in Simulink help you navigate your model:

- The Explorer bar lets you move up and down the hierarchy, or back and forth between different views in the Simulink Editor.
- The Model Browser provides a view of the **Model Hierarchy**, and lets you select and open different levels to quickly move through the hierarchy of the design.

Importing C/C++ Code as Custom Blocks

Introduction

Model Composer lets you import C or C++ code to create new blocks that can be added to a library for use in models along side other HLS Library blocks. This feature lets you build custom block libraries for use in Model Composer.

TIP: *The import function example in the product showcases most of the capabilities of importing C/C++ code using a series of small examples. Type `xmcOpenExample('import_function')` in the MATLAB command window to open the example.*

Using the xmcImportFunction Command

Model Composer provides the `xmcImportFunction` command, for use from the MATLAB command line, to let you specify functions defined in source and header files to import into Model Composer, and create Model Composer blocks, or block library. The `xmcImportFunction` command uses the following syntax:

```
xmcImportFunction('libName',{'funcNames'},'hdrFile',{'srcFiles'},
{'srchPaths'},'options')
```

Where:

- `libName`: A string that specifies the name of the Model Composer HLS library that the new block is added to. The library can be new, and will be created, or can be an existing library.
- `funcNames`: Specifies a list (cell array) of one or more function names defined in the source or header files to import as a Model Composer block. An empty set, {}, imports all functions defined in the specified header file (`hdrFile`). For functions inside namespaces, full namespace prefix needs to be given. For example, to import the function `'sinf'` in `hls_math`, the full function name `'hls::sinf'` needs to be specified.
- `hdrFile`: A string that specifies a header file (`.h`) which contains the function declarations, or definitions. This should be the full path to the header file if it is not residing inside the current working directory. For example, to import a function from `hls_math.h`, you need to specify the full path `'$XILINX_VIVADO/include/hls_math.h'`.

  IMPORTANT! *The function signature must be defined in the header file, and any Model Composer (XMC) pragmas must be specified as part of the function signature in the header file.*

- `srcFiles`: Specifies a list of one or more source files to search for the function definitions. When used in a model, the header and source files, together with the main header file of the model, `headerFile`, will be compiled into the shared library for simulation, and copied into the Target Directory specified for output generation in the Model Composer Hub block, as described in Adding the Model Composer Hub.

- `srchPaths`: Specifies a list of one or more search paths for header and source files. An empty set, {}, indicates no search path, in which case the code is looked for in the MATLAB current folder. Use `'$XILINX_VIVADO/include'` to include the HLS header files.

In addition, the `xmcImportFunction` command has the following `options`, which can follow the required arguments in any order:

- `'unlock'`: Unlock an existing library if it is locked. The `xmcImportFunction` command can add blocks to an existing library, but it must be unlocked in order to do so.
- `'override'`: Overwrite an existing block with the same name in the specified library.

TIP: *The help for `xmcImportFunction` can be accessed from the MATLAB command line, using `help xmcImportFunction`, and provides the preceding information.*

As an example, the following `simple.h` header file defines the `simple_add` function, with two double-precision floating point inputs and a pointer output. The function simply adds the two inputs and returns the sum as the output.

```
void simple_add(const double in1, const double in2, double *out) {
   *out = in1 + in2;
}
```

To import the `simple_add` function as a block in a Model Composer HLS library you can enter the following command at the MATLAB command prompt:

```
xmcImportFunction('SimpleLib',{'simple_add'},'simple.h',{},{})
```

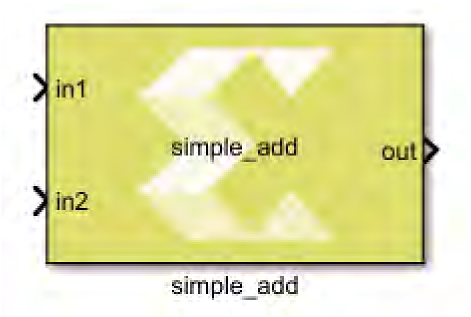
Where:

- `SimpleLib` is the name of the Model Composer HLS library to add the block to.
- `simple_add` is the function name to import.
- `simple.h` is the header file to look in.
- No C source files or search paths are specified. In this case, the function definition must be found in the specified header file, and only the MATLAB current folder will be searched for the specified files.

  TIP: *Model composer will give you a warning if you attempt to import a block with the same function name as a block that is already in the specified library.*

When `xmcImportFunction` completes, the `SimpleLib` library model will open with the `simple_add` block created as shown below.

Figure 180: **simple_add Block**



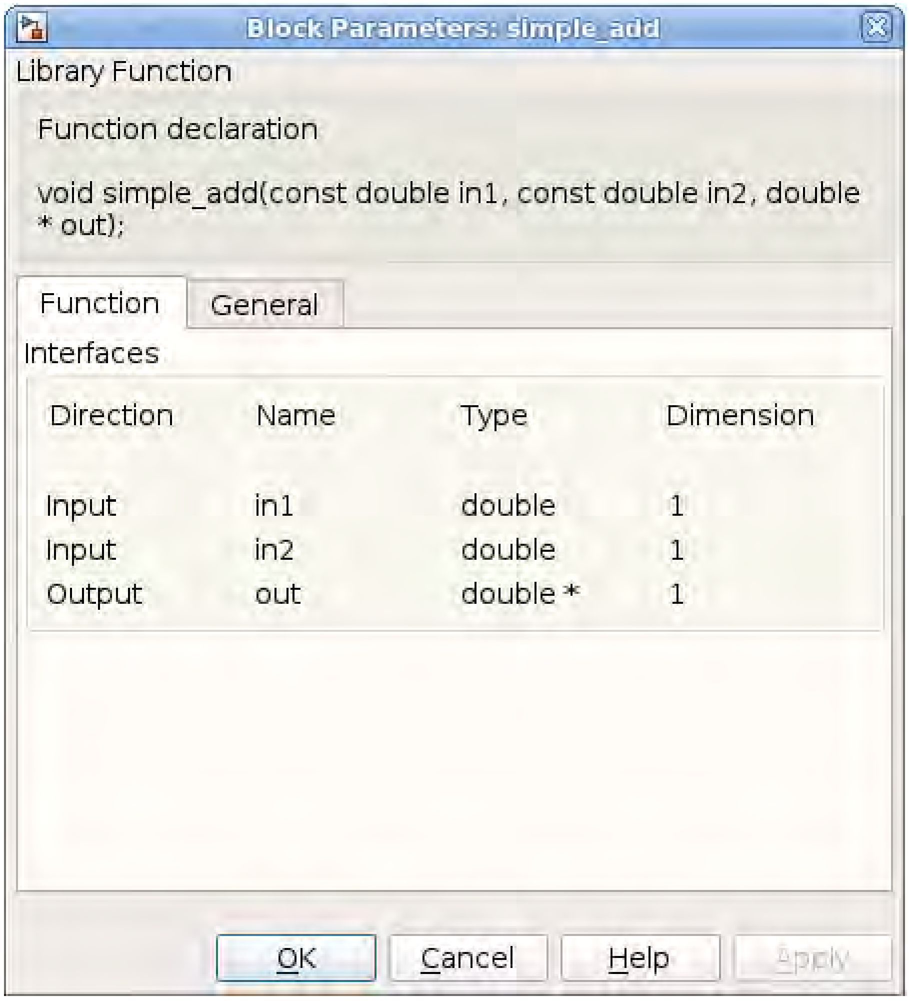
During simulation, the C/C++ code for the Library blocks will get compiled and a library file will get created and loaded. The library file will be cached to speed up initializing simulations on subsequent runs. You can change the source code underlying the library block without the need to re-import the block, or re-create the library, although the cached library file will be updated if you change the C/C++ source code.

However, if you change the function signature, or the parameters to the function, then you will need to rerun the `xmcImportFunction` command to recreate the block. In this case, you will also need to use the `override` option to overwrite the existing block definition in your library.

IMPORTANT! *You must rerun the `xmcImportFunction` command any time that you change the interface to an imported function, or the associated XMC pragmas. In this case, you should delete the block from your design and replace it with the newly generated block from the library. However, you can change the function content without the need to run `xmcImportFunction` again.*

After the block symbol has been created, you can double-click on the symbol to see the parameters of the imported block. You can quickly review the parameters as shown in the following figure to ensure the function ports have been properly defined.

Figure 181: **simple_add Block Parameters**



Block Parameters: simple_add

Library Function

Function declaration

void simple_add(const double in1, const double in2, double * out);

Function | General

Interfaces

Direction	Name	Type	Dimension
Input	in1	double	1
Input	in2	double	1
Output	out	double *	1

Importing C/C++ into Model Composer

While Model Composer lets you import C or C++ functions to create a library of blocks, it does have specific requirements for the code to be properly recognized and processed. The function source can be defined in either a header file (`.h`), or in a C or C++ source file (`.c`, `.cpp`), but the header file must include the function signature.

You can import functions with function arguments that are real or complex types of scalar, vectors, or matrices, as well as using all the data types supported by Model Composer HLS Library, including fixed-point data types. Model Composer also lets you define functions as templates, with template variables defined by input signals, or as customization parameters to be specified when the block is added into the model or prior to simulating the model. Using function templates in your code lets you create a Model Composer block that supports different applications, and can increase the re-usability of your block library. Refer to Defining Blocks Using Function Templates for more information.

IMPORTANT! *The function signature must be defined in the header file, and any Model Composer (XMC) pragmas must be specified as part of the function signature in the header file.*

The `xmcImportFunction` command supports C++ functions using `std::complex<T>` or `hls::x_complex<T>` types. For more details, see the explanation in Using Complex Types.

If the input of an imported function is a 1-D array, the tool can perform some automatic mappings between the input signal and the function argument. For example, if the function argument is `a[10]`, then the connected signal in Model Composer can be either a vector of size 10, or a row or column matrix of size 1x10 or 10x1.

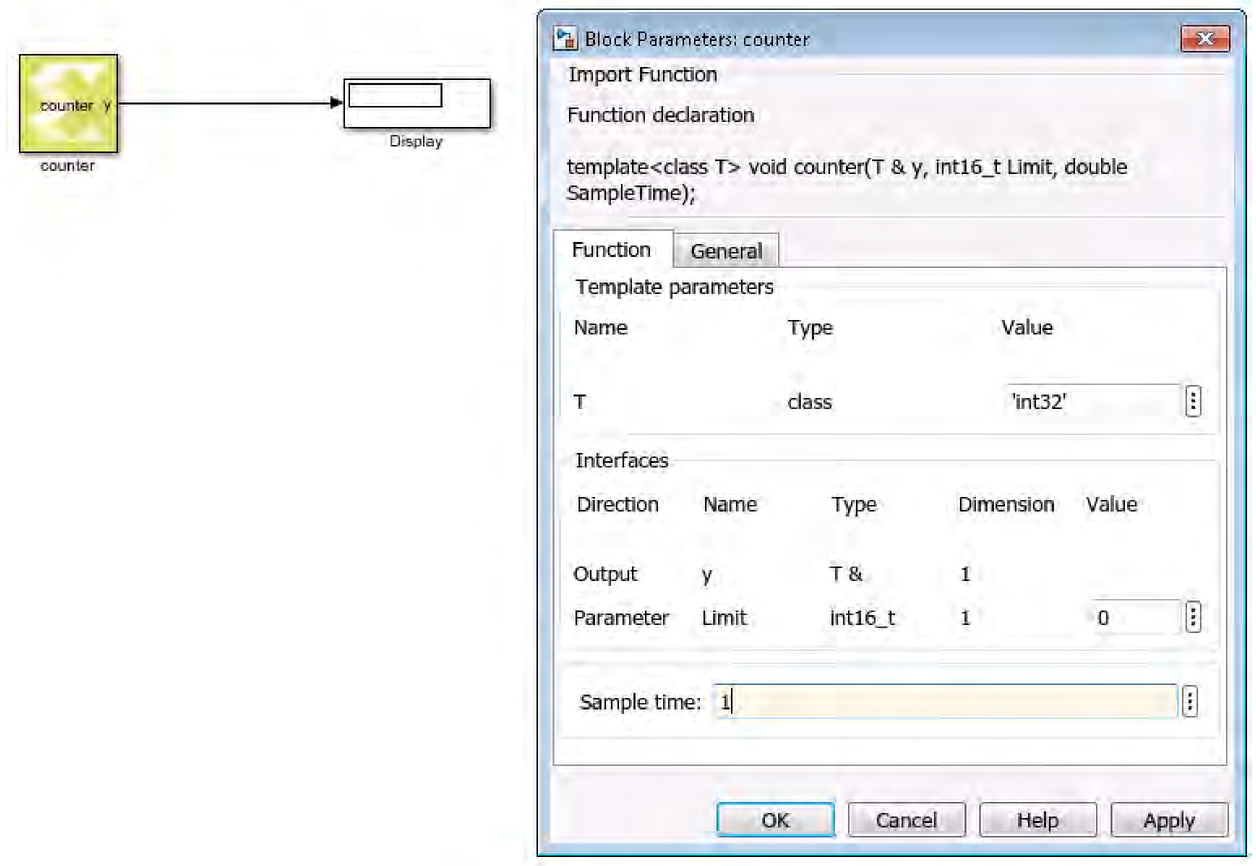
However, if all of the inputs and outputs of an imported function are scalar arguments, you can connect a vector signal, or a matrix signal to the input. In this case, the imported function processes each value of the vector, or matrix on the input signal as a separate value, and will combine those values into the vector, or matrix on the output signal. For example, a vector of size 10 connected to a scalar input, will have each element of the vector processed, and then returned to a vector of size 10 on the output signal.

You can import functions that do not have any inputs, and instead only generate outputs. This is known as a source block, and can have an output type of scalar, vector, complex, or matrix. You can also import source blocks with multiple outputs. The following example function has no input port, and `y` is the output:

```
#include <stdint.h>
#include <ap_fixed.h>

#pragma XMC OUTPORT y
#pragma XMC PARAMETER Limit
template <typename T>
void counter(T &y, int16_t Limit)
{
    static T count = 0;

    count++;

    if (count > Limit)
            count =0;
    y = count;
}
```

TIP: *Because source blocks have no inputs, the `SampleTime` parameter is automatically added when the block is created with `xmcImportFunction` command, as shown in the Function declaration in the following image. The default value is -1 which means the sample time is inherited from the model. You can also explicitly specify the sample time by customizing the block when it is added to a model, as shown below.*

Figure 182: **Setting Sample Time for a Source Block**



Block Parameters: counter

Import Function

Function declaration

template<class T> void counter(T & y, int16_t Limit, double SampleTime);

Function | General

Template parameters

Name	Type	Value
T	class	'int32'

Interfaces

Direction	Name	Type	Dimension	Value
Output	y	T &	1	
Parameter	Limit	int16_t	1	0

Sample time: 1

OK | Cancel | Help | Apply

The direction of ports for the function arguments can be determined automatically by the `xmcImportFunction` command, or manually specified by pragma with the function signature in the header file.

- Automatically determining input and output ports:
  - The `return` value of the function is always defined as an output, unless the return value is `void`.
  - A formal function argument declared with the `const` qualifier is defined as an input.
  - An argument declared with a reference, a pointer type, or an array type without a `const` qualifier is defined as an output.
  - Other arguments are defined as inputs by default (e.g., scalar read-by-value).
- Manually defining input and output ports:
  - You can specify which function arguments are defined as inputs and outputs by adding the INPORT and OUTPORT pragmas into the header file immediately before the function declaration.
  - `#pragma XMC INPORT <parameter_name> [, <parameter_name>...]`

- `#pragma XMC OUTPORT <parameter_name> [, <parameter_name>...]`

In the following example `in` is automatically defined as an input due to the presence of the `const` qualifier, and `out` is defined as an output. The imported block will also have a second output due to the integer `return` value of the function.

```
int func(const int in, int &out);
```

In the following function `in` is automatically defined as an input, and `out` as an output, however, there is no `return` value.

```
void func(const in[512], int out[512]);
```

In the following example the ports are manually identified using pragmas that have been added to the source code right before the function declaration. This is the only modification to the original C++ code needed to import the function into Model Composer. In this example the pragmas specify which parameter is the input to the block and which parameter is the output of the block.

```
#pragma XMC INPORT din
#pragma XMC OUTPORT dout
void fir_sym (ap_fixed<17,3,AP_TRN,AP_WRAP> din[100],
              ap_fixed<17,3,AP_TRN,AP_WRAP> dout[100]);
```

TIP: *`ap_fixed` specifies a fixed-point number compatible with Vitis HLS.*

Manually adding pragmas to the function signature in the header file to define the input and output parameters of the function is useful when your code does not use the `const` qualifier, and adding the `const` qualifier can require extensive editing of the source code when there is a hierarchy of functions. It also makes the designation of the inputs and outputs explicit in the code, which can make the relationship to the imported block more clear.

Some final things to consider when writing C or C++ code for importing into Model Composer:

- You should develop your source code to be portable between 32 bit and 64 bit architectures.
- Your source code can use Vitis HLS pragmas for resource and performance optimization, and Model Composer uses those pragmas but does not modify or add to them.
- If your code has static variables, the static variable will be shared across all instances of the blocks that import that function. If you do not want to share that variable across all instances you should copy and rename the function with the static variable and import a new library block using the `xmcImportFunction` command.

- If you use C (`.c`) source files to model the library function (as opposed to C++ (`.cpp`) source files), the `.h` header file must include an `extern "C"` declaration for the downstream tools (such as Vitis HLS) to work properly. An example of how to declare the `extern "C"` in the header files is as follows:

```
// c_function.h:
#ifdef __cplusplus
extern 'C' {
#endif
void c_function(int in, int &out);
#ifdef __cplusplus
}
#endif
```

Using Complex Types

C++ code for Vitis HLS can use `std::complex<T>` or `hls::x_complex<T>` to model complex signals for xmcImportFunction blocks.

The code generated by Model Composer uses `std::complex` for representing complex signals. If your imported C/C++ block function is modeled with `hls::x_complex`, when generating the output code Model Composer will automatically insert `std::complex`-to-`hls::x_complex` adapters to convert the complex types for the block ports.

The C++ code must also include the required header file for the complex type declaration. For the `hls::x_complex` type, the `xmcImportFunction` command must also include `‘$XILINX_VIVADO/include’` search path for the `hls_x_complex.h` header file. For example, the following imports the `complex_mult` function, and specifies the needed include path:

```
xmcImportFunction('my_lib',{'complex_mult'}, 'complex_mult.h', {},
{'$XILINX_VIVADO/include'});
```

Example Functions Using Complex Types

```
#include "hls_x_complex.h"
hls::x_complex<double>
complex_mult(hls::x_complex<double> in1, hls::x_complex<double> in2)
{ return in1 * in2; }

#include <complex>
std::complex<double>
complex_mult2(std::complex<double> in1, std::complex<double> in2)
{ return in1 * in2; }

#include <complex>
void
complex_mult3(std::complex<double> in1, std::complex<double> in2,
std::complex<double> &out1)
{ out1.real(in1.real() * in2.real() - in1.imag() * in2.imag());
   out1.imag(in1.real() * in2.imag() + in1.imag() * in2.real()); }
```

Defining Blocks Using Function Templates

IMPORTANT! *To use template syntax, your function signature and definition should both be specified in a header file when running `xmcImportFunction`.*

While it is common to write functions that accept only a predetermined data type, such as `int32`, in some cases you may want to create a block that accepts inputs of different sizes, or supports different data types, or create a block that accepts signals with different fixed-point lengths and fractional lengths. To do this you can use a function template that lets you create a block that accepts a variable signal size, data type, or data dimensions.

You can define blocks using function templates, as shown in the following example:

```
#include <stdint.h>
template <int ROWS, int COLS>
void simple_matrix_add(const int16_t in1[ROWS][COLS],
                       const int16_t in2[ROWS][COLS],
                       int16_t out[ROWS][COLS]) {
   for (int i = 0; i<ROWS; i++) {
      for (int j = 0; j<COLS; j++) {
         out[i][j] = in1[i][j] + in2[i][j];
      }
   }
}
```

The example uses the template parameters `ROWS` and `COLS`. The actual dimensions of the input and output arrays, `in1[ROWS][COLS]` for instance, are determined at simulation time by the dimensions of the input signals to the block. `ROWS` and `COLS` are template parameters used to define the dimensions of the function arguments, and also used in the body of the function, `i<ROWS` for example.

Use the command below to import the function into Model Composer:

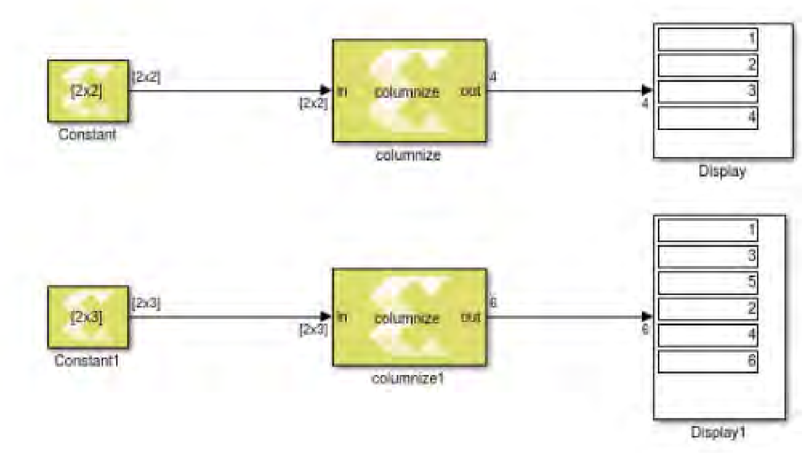
```
xmcImportFunction('SimpleLib',{'simple_matrix_add'},...
'template_example.h',{},{},'unlock')
```

TIP: *In the example above the ellipsis (...) is used to indicate a continuation of the command on the next line. Refer to Continue Long Statements on Multiple Lines in the MATLAB documentation for more information.*

You can perform simple arithmetic operations using template parameters. For example, the following code multiplies the ROWS and COLS of the input matrix to define the output, as shown in the figure below.

```
#include <stdint.h>
#pragma XMC INPORT in
#pragma XMC OUTPORT out
template<int ROWS,int COLS>
void columnize(const int16_t in[ROWS][COLS], int16_t out[ROWS*COLS]) {
   for (int i = 0; i<ROWS; i++) {
        for (int j = 0; j<COLS; j++) {
            out[i*COLS+j] = in[i][j];
        }
    }
}
```

Figure 183: **Columnize Function**



Other simple supported operations include +, -, *, /, %, <<, and >>, using both the template parameters and integer constants. For example:

```
template<int M, int N>
void func(const int in[M][N], int out[M*2][M*N]);

template<int ROWS, int COLS>
void func(array[2 * (ROWS + 1) + COLS + 3]);
```

You can also define a function template that uses a fixed-point data type of variable word length and integer length using function templates, as shown in the following example:

```
#include <stdint.h>
#include <ap_fixed.h>
#pragma XMC OUTPORT out
template <int WordLen, int IntLen>
void fixed_add(const ap_fixed<WordLen,IntLen> in1,
                      const ap_fixed<WordLen,IntLen> in2,
                      ap_fixed<WordLen+1,IntLen> &out) {
   out = in1+in2;
}
```

The example above uses the fixed point notations from Vitis HLS, which specifies the word length and the integer length. In Model Composer, as described in Working with Data Types, you specify the word length and the fractional length. This requires you to use some care in connecting fixed point data types in Model Composer to the imported `fixed_add` block. For example, in the function above if `WordLen` is 16 and `IntLen` is 11, in Model Composer fixed point data type the word length is 16, and the fractional length is 5. For more information on fixed-point notation in Vitis HLS, refer to the *Vitis High-Level Synthesis User Guide* (UG1399).

TIP: *As shown in the example above, simple arithmetic operations are also supported in the fixed point template parameter.*

To import the `fixed_add` function and create a block in Model Composer, use the following command:

```
xmcImportFunction('SimpleLib',{'fixed_add'},fixed_example.h',{},...
{'$XILINX_VIVADO/include'})
```

Function Templates for Data Types

Function templates for data types are functions that can operate with generic data types. This lets you create library functions that can be adapted to support multiple data types without needing to replicate the code or block in the Model Composer HLS block library to support each type. The `xmcImportFunction` command in Model Composer will create generic library blocks which the user of the block can connect to signals of any data types supported by the block.

The data type (`typename`) template parameters are resolved at simulation run time, when the code and simulation wrapper are generated. The parameters are replaced during simulation by the actual data types that are specified by the signals connecting to the library block. The resolved data types can only be the types that Model Composer supports as discussed in Working with Data Types.

To import a block that accepts multiple data types, you use a function template. For example:

```
template <typename T>
T max(T x, T y) {
   return (x > y) ? x : y;
}
```

Or, in the case of a function with complex function arguments, the example would appear as as follows:

```
#include <complex>
template <typename T>
void mult_by_two(std::complex< T > x, std::complex< T > *y)
{
   *Out = In1 * 2;
}
```

The determination of the type is made by Model Composer during simulation. The `typename` (or `class`) parameters are propagated from input signals on the block, or are customization parameters that must be defined by the user at simulation run time.

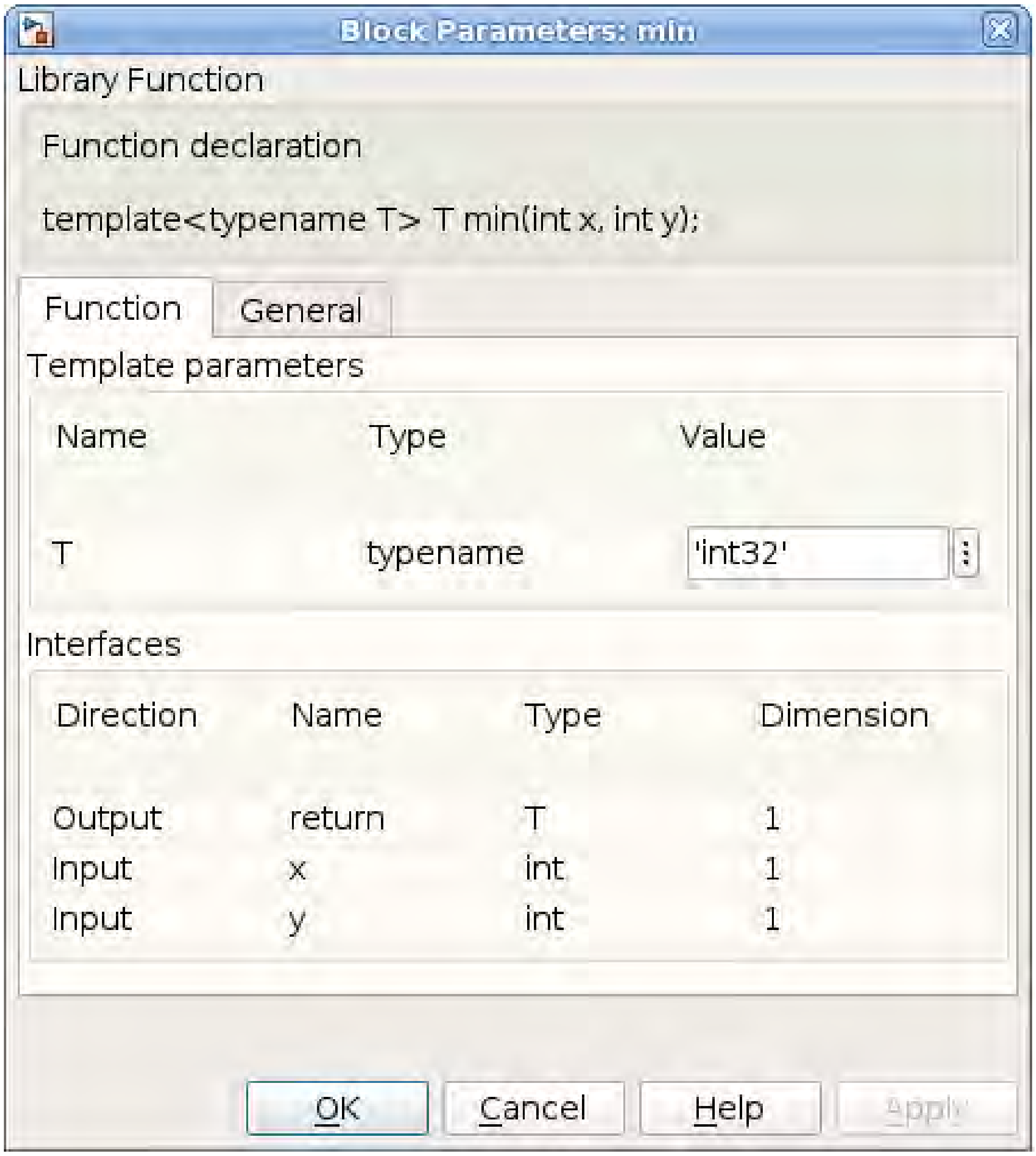
IMPORTANT! *The data type for a function or class cannot be propagated from an output.*

For example, the following function template specifies the parameter ‘T’ as a customization parameter. Because it is not associated with either input argument, 'x' or 'y', it must be specified by the user when the block is added to the model:

```
template <typename T>
T min(int x, int y) {
   return (x < y) ? x : y;
}
```

The Block Parameters dialog box for the generated Library Function block has an edit field to enter the template argument as shown in the following figure.

Figure 184: **Library Function Block Parameters**



Library Function

Function declaration

template<typename T> T min(int x, int y);

Template parameters

Name	Type	Value
T	typename	'int32'

Interfaces

Direction	Name	Type	Dimension
Output	return	T	1
Input	x	int	1
Input	y	int	1

In the template syntax, the data type template parameters for function or class can be specified with other template parameters. The order of specification is not important. For example:

```
template <typename T1, int ROWS, int COLS, int W, int I>
T1 func(T1 x[ROW][COLS], ap_fixed<W, I> &y) {
...
}
```

IMPORTANT! *In the example above, notice that the 'T1' template parameter is used to specify both the function return and the data type of the input 'x'. In this case, because it is a single template parameter, both arguments will resolve to the same data type that is propagated from the input signal to the block.*

SUPPORTED_TYPES/UNSUPPORTED_TYPES Pragma

When defining a data type (`typename`) template parameter (or `class`), you can also define the accepted data types for the variable by using either the `SUPPORTED_TYPES` or `UNSUPPORTED_TYPES` pragma as part of the function signature. This is shown in the following code example.

```
#pragma XMC INPORT x
#pragma XMC INPORT y
#pragma XMC SUPPORTED_TYPES T: int8, int16, int32, double, single, half
template <class T>
T max(T x, T y) {
   return (x > y) ? x : y;
}

#pragma XMC UNSUPPORTED_TYPES T: boolean
#pragma XMC INPORT x, y
template <typename T>
T min(T x, T y) {
   return (x < y) ? x : y;
}
```

Model Composer supports an extensive list of data types as discussed in Working with Data Types. To specify which of these data types the template parameter supports, you can either include the list of supported types, or the unsupported types. The `SUPPORTED_TYPES` and `UNSUPPORTED_TYPES` pragmas are simply two opposite views of the same thing:

- `SUPPORTED_TYPES`: Specifies a template parameter name (`param`), and the list of data types that are accepted by that parameter. This implies the exclusion of all types not listed.

  ```
  #pragma XMC SUPPORTED_TYPES param: type1, type2, ...
  ```

- `UNSUPPORTED_TYPES`: Specifies a template parameter name (`param`), and the list of data types that are not accepted by that parameter. This implies the inclusion of all types not listed.

  ```
  #pragma XMC UNSUPPORTED_TYPES param: type1, type2, ...
  ```

With the `SUPPORTED_TYPES` or `UNSUPPORTED_TYPES` pragma in place, Model Composer will check the type of input signal connected to the block to ensure that the data type is supported. Without the use of one of these pragmas, the data type template parameter will accept any of the data types supported by Model Composer.

Function Template Specialization and Overloading

Specialization is supported for function templates in the `xmcImportFunction` command. Model Composer will create the library block for the generic function template, supporting multiple data types, but the block will also include any specialized functions to be used when connected to input signals with matching data types. Both the generic function, and any specialization functions are compiled into the block DLL. For example:

```
template <typename T>
T min(T x, T y) {
   return (x < y) ? x : y;
}

template <>
bool min<bool>(bool x, bool y) {
...
}
```

In this case, Model Composer will call the specialized boolean form of the `min` function when the block is connected to boolean signals.

Overloading of a function with the same number of input/output arguments is also supported by Model Composer. For example, the following defines two forms of the function:

```
int func(int x);
float func(float x);
```

You can also overload a function template as shown below:

```
template <typename T>
int func(int x, T y);

template <typename T>
float func(float x, T y);
```

TIP: *Overloading functions with different numbers of input/output arguments or different argument dimensions is not supported, and must be defined as separate functions.*

Defining Customization Parameters

Template parameters can be used to define the port array sizing and data type, and the parameters are defined by the input signals on the block. Additional customization parameters can also be defined, which are not defined by input signals, and thus must be defined by the user using the Block Parameter dialog box sometime before simulation run time.

There are two methods to define customization parameters for a library function block:

- Using C/C++ function templates.

- Assigning the Model Composer (XMC) PARAMETER pragma to a function argument, defining it as not connecting to an input or output of the block, but rather as a customization parameter.

There are pros and cons to both methods, which are discussed below.

Function Templates

The first method, defines a function template that uses template parameters for customization. A template parameter defines a customization parameter in Model Composer if its value is not determined by an input signal, or derived by the function as an output. You can use customization parameters to define the values, data types, or data dimensions of output ports, or for parameters in use in the function body.

IMPORTANT! *The template function signature and function definition must be defined in the C/C++ header file.*

The template parameter for the function argument is defined using standard function template syntax, but the template parameter is not assigned to an input argument in the function signature. When the block is instantiated into a model, Model Composer identifies template parameters whose values are not determined by input signals, and lets the user define the values for those customization parameters. Values can be defined for customization parameters in the model any time prior to simulation.

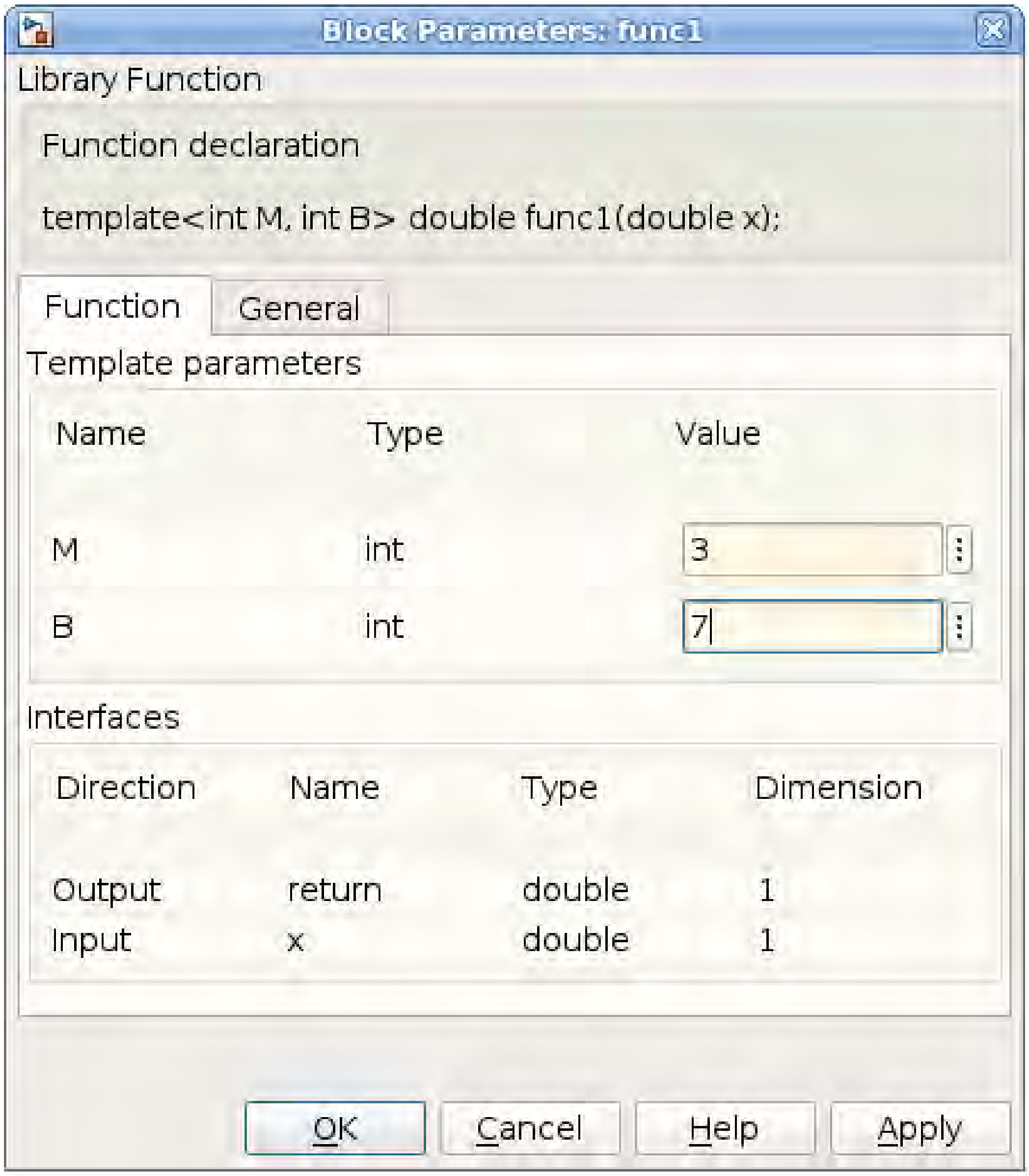
For function templates, the customization parameters can only be integer values to define the size or dimensions of a data type, or can only be scalar variables with definable data types. Model Composer defines a default value of 0 for integer parameters, and 'int32' for data type, or typename parameters.

In the function template example below, the template parameters ‘M’ and 'B' define customization parameters because the parameter values are not inherited from the input signal to the block. In this case, the parameters need to be customized by the user when the block is added to the model, or any time before simulation.

```
template <int M, int B>
double func1(double x) {
   return x * M + B;
}
```

Customization parameters are displayed in the Block Parameters dialog box for the imported block as shown for the `func1` function below. Double click on a block in the model to open the **Block Parameters** dialog box, then enter the value for any editable parameters, such as 'M' and 'B' below.

Figure 185: **Entering Parameter Values**



Library Function

Function declaration

template<int M, int B> double func1(double x);

Function | General

Template parameters

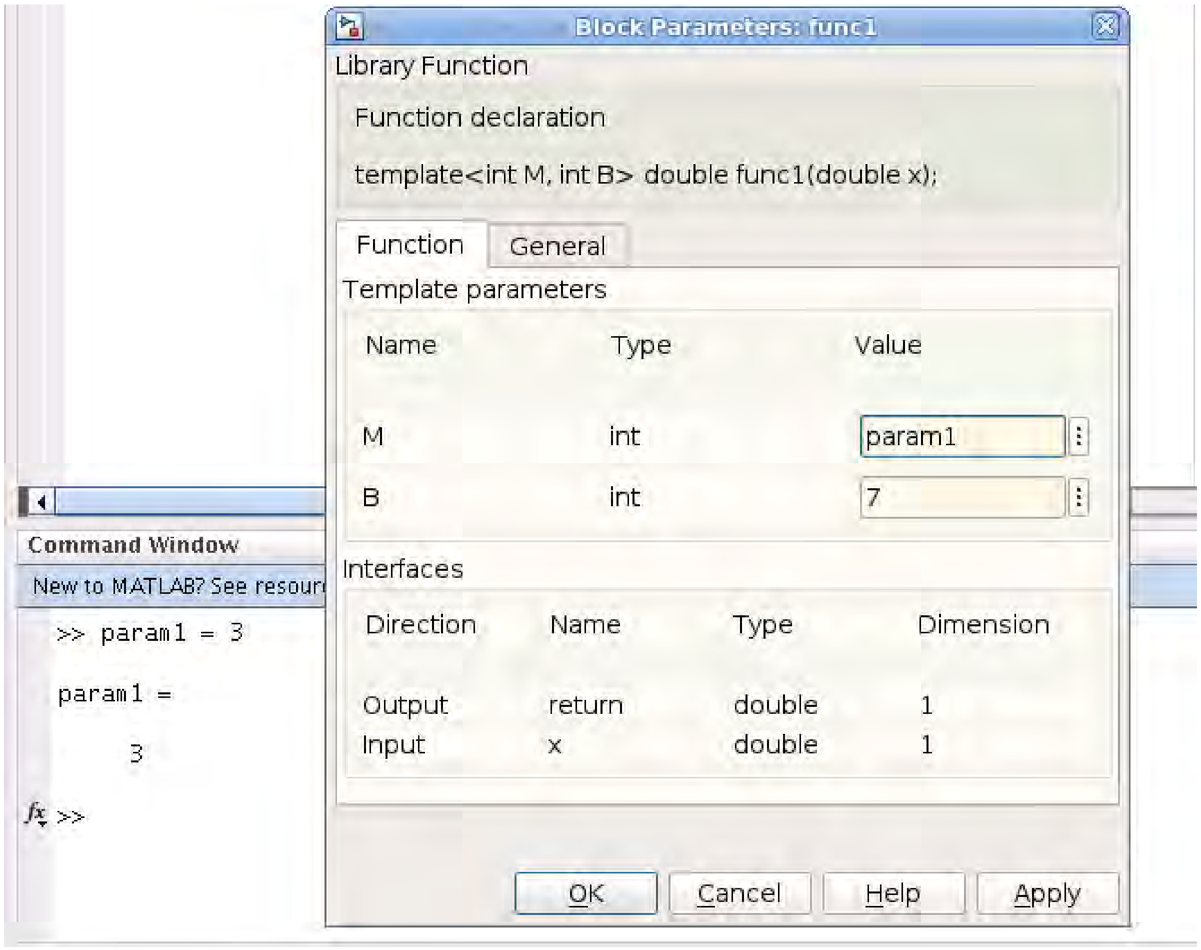
Name	Type	Value
M	int	3
B	int	7

Interfaces

Direction	Name	Type	Dimension
Output	return	double	1
Input	x	double	1

Optionally, the user can also specify the name of a MATLAB workspace variable in the text field for the customization parameter, and have the value determined by Model Composer through the MATLAB variable. For example, the variable `param1` is defined in the MATLAB workspace, and used to define the value for 'M'.

Figure 186: **Defining Parameters using Workspace Variables**



Name	Type	Value
M	int	param1
B	int	7

Direction	Name	Type	Dimension
Output	return	double	1
Input	x	double	1

```
>> param1 = 3

param1 =

     3

>>
```

PARAMETER Pragma

The second method defines function arguments as customization parameters through the use of the Model Composer `PARAMETER` pragma.

To declare that a function argument is a customization parameter, you must add the `PARAMETER` pragma with the parameter name, or list of names, before the function signature in the header file. You can specify multiple parameters with one pragma, or have separate pragmas for each, as shown below.

```
#pragma XMC PARAMETER <name1>, <name2>
#pragma XMC PARAMETER <name3>
function declaration(<name1>, <name2>, <name3>)
```

When a function argument is declared a customization parameter by pragma, the `xmcImportFunction` command will not create an input or output port on the block for that argument. It will be defined for use inside the function body only. When the block is added to a model, a customization field is added to the Block Parameter dialog box, and the user of the block can define values for the customization parameters.

Using the `PARAMETER` pragma on a function argument that is already driven by the input signal will be flagged as an error or a warning. In this case, the signal input propagation through the function will have higher precedence than the customization parameter.

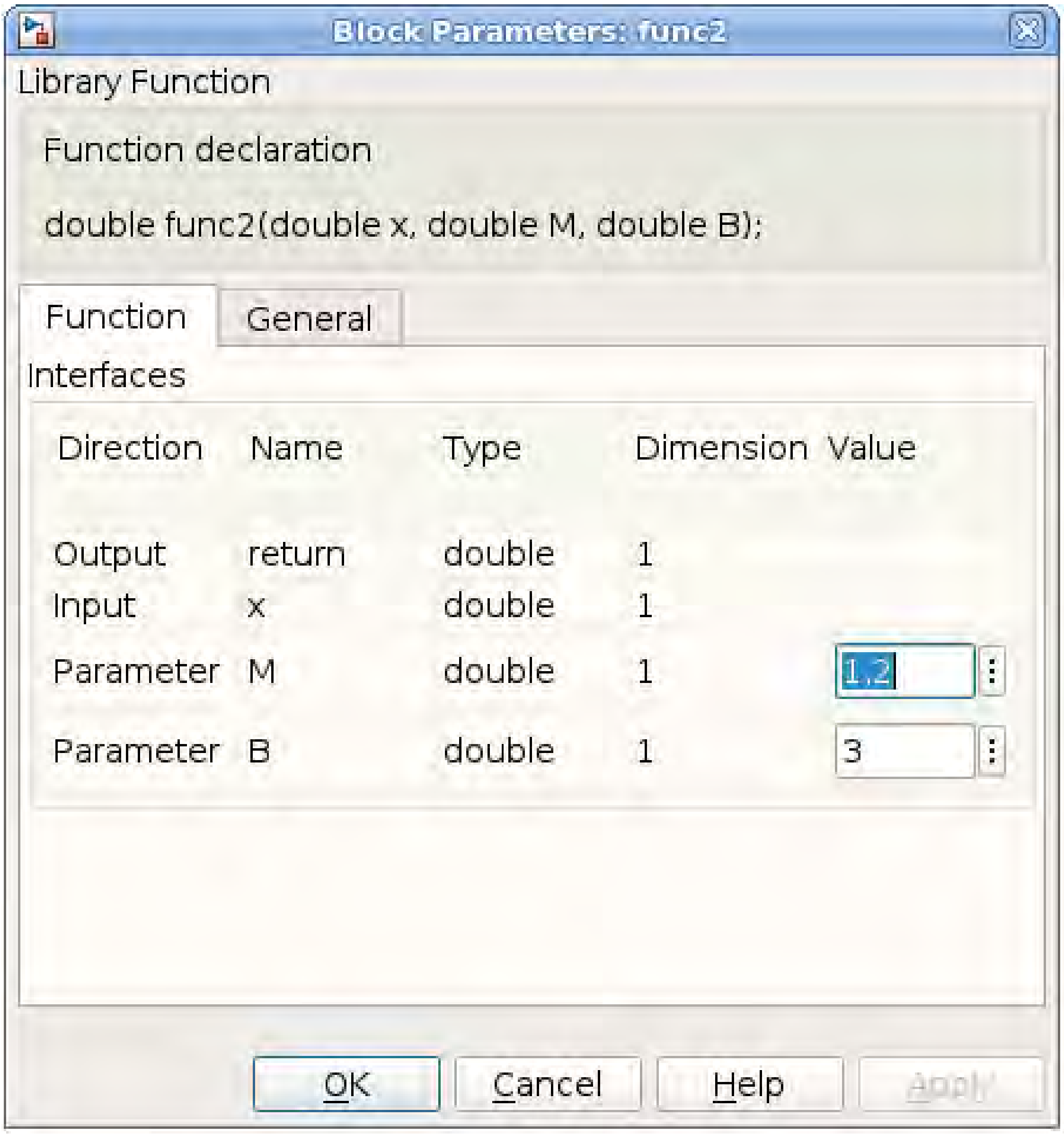
While the function templates method only supports scalar and integer type customization parameters, the `PARAMETER` pragma supports integer, floating point or fixed point data type for the parameters. The customization parameters also can be scalar, vector or a two-dimensional matrix. In addition, while the function template defines default values of 0 for integer types, and `int32` for the data type, the `PARAMETER` pragma lets you define default value for the parameters. Model Composers defines default values of 0 for all parameters that do not have user-defined defaults.

The example below uses the Model Composer `PARAMETER` pragma to define the customization parameters 'M' and 'B'.

```
#pragma XMC PARAMETER M, B
double func2(double x, double M = 1.2, double B = 3) {
return x * M + B;
}
```

The 'M' and 'B' customization parameters also have default values assigned: `M=1.2`, `B=3`. The default values for the customization parameters are assigned to the arguments in the function signature, and are displayed in the **Block Parameters** dialog box when opened for edit, as shown below.

Figure 187: **Customization Parameters with Defaults**



Function declaration

double func2(double x, double M, double B);

Direction	Name	Type	Dimension	Value
Output	return	double	1	
Input	x	double	1	
Parameter	M	double	1	1,2
Parameter	B	double	1	3

IMPORTANT! *If you define default values for the customization parameters of any argument, the C/C++ language requires that all arguments following that one must also have default values assigned, because the function can be called without arguments having default values. Therefore, you should add all customization parameters with default values at the end of the function argument list.*

Vector and Matrix Customization Parameters

The PARAMETER pragma method can also be used to specify customization parameters with vector and matrix dimensions, or values. In the following example the `coef` vector is defined by the pragma as a customization parameter:

```
#pragma XMC PARAMETER coef
#pragma XMC INPORT din
#pragma XMC OUTPORT dout
#pragma XMC SUPPORTS_STREAMING
void FIR(ap_fixed<17, 3> din[100], ap_fixed<17, 3> dout[100],
ap_fixed<16, 2> coef[52]);
```

The constant array values of the customization parameter are entered in MATLAB expression format. Note that commas are optional:

- Vector parameter: `[val1, val2, val3, ...]`
- Matrix parameter (row-major order): `[val11, val12, val13, ...; val21, val22, val23, ...; ...]`

Interface Output Types and Sizes

Customization parameters can also be used to directly set the data types and dimension size for output ports whose values are not determined by inputs to the function. In the function below, the template variables define the word length and fractional length of the `ap_fixed` data type and the array size.

```
template <typename T1, int N1, int W2, int I2, int N2>
void func(const T1 in[N1], ap_fixed<W2, I2> out[N2]) {
...
}
```

The template variables 'W2, 'I2'' and 'N2' define customization parameters because the values must be set by the user rather than determined from the input arguments. However, Model Composer recognizes that the template variables 'T1' and 'N1' are specified on the input port, and so the data type (`typename`) and the size of the input vector are not customization parameters, but rather get defined by the input signal on the block.

To set the data type for output ports, or arguments used in the body of the function, the `typename` specified must be one of the Model Composer supported data types, including the signed or unsigned fixed data types.

Table 11: **Model Composer Supported Data Types**

Supported Typenames
'int8'
'uint8'
'int16'

Table 11: **Model Composer Supported Data Types** *(cont'd)*

Supported Typenames
'uint16'
'int32'
'uint32'
'double'
'single'
'x_half'
'boolean'
'x_sfix<n1>_En<n2>'
'x_ufix<n1>_En<n2>'

In the example function below, while the `typename` for 'T1' is determined by the input signal, you can set the `typename` for 'T2' in the **Block Parameters** dialog box on the mask, when the block is added to a model, or before simulation run time:

```
template <typename T1, int N1, typename T2, int N2>
void func(const T1 in[N1], T2 out[N2]) {
...
}
```

Pragmas for xmcImportFunction

XMC SUPPORTS_STREAMING

The Model Composer `SUPPORTS_STREAMING` pragma indicates that the array parameters of a function support streaming data. This means that each element of the array is accessed once in a strict sequential order and indicates to Model Composer to optimize the design for streaming data. There can be no random access to the non-scalar arguments of a function to which the `SUPPORTS_STREAMING` pragma is applied.

The following example illustrates the difference between random access and sequential access. The `transform_matrix` output array of the `create_transform_matrix` function is addressed in a random order. It accesses the last row of the `transform_matrix` first, followed by the first and second row. This prevents the block from supporting streaming data.

```
void create_transform_matrix(const float angle, const float center_x,
                             const float center_y, float transform_matrix[3]
[3]) {
    float a = hls::cosf(angle);
    float b = hls::sinf(angle);

    transform_matrix[2][0] = 0;
    transform_matrix[2][1] = 0;
    transform_matrix[2][2] = 0;
    transform_matrix[0][0] = a;
    transform_matrix[0][1] = b;
    transform_matrix[0][2] = (1-a)*center_x-b*center_y;

    transform_matrix[1][0] = -b;
    transform_matrix[1][1] = a;
    transform_matrix[1][2] = b*center_x +(1-a)*center_y;
}
```

To change this function to support streaming data, it can be modified as depicted below to address the `transform_matrix` output array in a sequential manner:

```
#pragma XMC SUPPORTS_STREAMING
void create_transform_matrix(const float angle, const float center_x,
                             const float center_y, float transform_matrix[3]
[3]) {
    float a = hls::cosf(angle);
    float b = hls::sinf(angle);

    transform_matrix[0][0] = a;
    transform_matrix[0][1] = b;
    transform_matrix[0][2] = (1-a)*center_x-b*center_y;

    transform_matrix[1][0] = -b;
    transform_matrix[1][1] = a;
    transform_matrix[1][2] = b*center_x +(1-a)*center_y;

    transform_matrix[2][0] = 0;
    transform_matrix[2][1] = 0;
    transform_matrix[2][2] = 0;
}
```

As shown in the preceding example, to specify that an imported function supports streaming, simply add the `SUPPORTS_STREAMING` pragma in the C or C++ header file before the function declaration:

```
#pragma XMC SUPPORTS_STREAMING
```

If the function has array arguments that are accessed sequentially, but `SUPPORTS_STREAMING` is not specified, then a subsystem using that block will not be implemented in a streaming architecture. This means the performance of the function will not be optimized.

IMPORTANT! *If your function accesses the array arguments in random order, you must not specify the `SUPPORTS_STREAMING` pragma or an error will be returned when generating output or verifying your design.*

The following is an example of a function that accesses the array arguments in a strictly sequential order, supporting the streaming of data. This function flips the rows of an input image horizontally. The function accesses the input image in a sequential order and buffers two rows of the input image in a circular buffer. Once two full rows are buffered, the function writes the buffer content to the function argument in a sequential order. As such, this function supports streaming, and uses the `SUPPORTS_STREAMING` pragma to specify it.

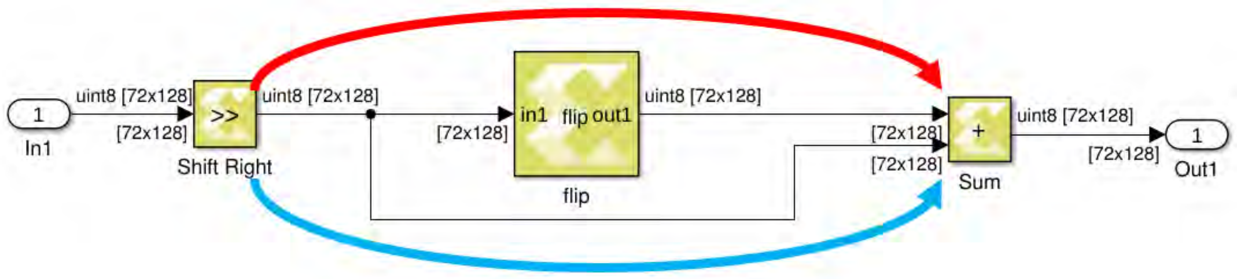
```
#ifndef _MY_FUNCS
#define _MY_FUNCS

#include <stdint.h>

#pragma XMC INPORT in1
#pragma XMC OUTPORT out1
#pragma XMC SUPPORTS_STREAMING
#pragma XMC BUFFER_DEPTH 4+2*WIDTH
// This function reverses each of the rows of the input image.
template<int WIDTH, int HEIGHT>
void
flip(uint8_t in1[HEIGHT][WIDTH],
     uint8_t out1[HEIGHT][WIDTH])
{
#pragma HLS dataflow

   uint8_t buf[2][WIDTH];

   int readBuf = 0;
   int writeBuf = 0;
   for (int row = 0; row < HEIGHT + 2; ++row) {
      for (int col = 0; col < WIDTH; ++col) {
#pragma HLS DEPENDENCE array inter false
#pragma HLS PIPELINE II=1
         if (row < HEIGHT) {
            buf[writeBuf][col] = in1[row][col];
            if (col == WIDTH-1) {
               ++writeBuf;
               writeBuf = (3 == writeBuf) ? 0 : writeBuf;
            }
         }
         if (row > 1) {
            out1[row - 2][col] = buf[readBuf][WIDTH- 1 - col];
            if (col == WIDTH-1) {
               ++readBuf;
               readBuf = (3 == readBuf) ? 0 : readBuf;
            }
         }
      }
   }
}
#endif
```

XMC BUFFER_DEPTH

The Model Composer `BUFFER_DEPTH` pragma provides information for properly sizing the buffers that connect the blocks in an implementation. These buffers are implemented as FIFOs in hardware. By default, Model Composer sets the depths of these buffers to 1. However, if your design has re-convergent paths (two paths converging into the same node) and the processing of data from the blocks of one path are not in lockstep with the processing of data from the other path, then a deadlock can occur. To avoid the deadlock the depth of one or more of the buffers on the paths can be increased to store the data. The following example illustrates this concept.

In the following diagram the `Sum` block consumes both the output signal of the `flip` block (red path), and the output of the `Shift Right` block (blue path). The `flip` block has been created with the `xmcImportFunction` command, and its source code is shown in the `flip` function previously described.

Figure 188: **Buffer Depth**



From the code for the `flip` block, you can see that the block needs to read 2 full rows before producing the first output. If the `BUFFER_DEPTH` pragma is not specified for the block, Model Composer sets the buffer sizes to 1 for the signals in the diagram. This results in deadlock, because the `flip` block reads 257 pixels from the input FIFO before producing the first output. However, by default, the parallel blue path feeding the second input of `Sum` has only enough storage for 1 pixel.

TIP: *Vitis HLS provides some capability to detect deadlocks during C/RTL co-simulation. In case a deadlock is detected, the tool prints out messages showing which FIFOs are involved in the deadlock, to help identify FIFOs that may require a BUFFER_DEPTH of more than 1.*

To change the default BUFFER_DEPTH, as shown in the `flip` function, place the pragma in the header file before the function declaration:

```
#pragma XMC BUFFER_DEPTH <depth>
```

Where `<depth>` specifies the buffer depth, and can be specified as a value or an expression.

By specifying `#pragma XMC BUFFER_DEPTH 4+2*WIDTH` in the `flip` function, Model Composer can determine that there is an imbalance in processing among the re-convergent paths, and address this imbalance by setting the buffer depth for the second input to the `Sum` block (blue path) to match the buffer depth of the `flip` block.

TIP: *Determining the minimal buffer depth may require a bit of trial and error because it also depends on the timing of the reads and writes into the FIFOs in the RTL code. In the `flip` function example, 256 (or `2*WIDTH`) was not sufficient `BUFFER_DEPTH`, but 260 (or `4+2*WIDTH`) prevented the deadlock.*

XMC THROUGHPUT_FACTOR

The Model Composer `THROUGHPUT_FACTOR` pragma provides some control over the throughput of an `xmcImportFunction` block. You can add the `THROUGHPUT_FACTOR` pragma to your function header file, along with the `SUPPORTS_STREAMING` pragma as shown in the following example:

```
#pragma XMC THROUGHPUT_FACTOR TF_param: 1,2,4
#pragma XMC SUPPORTS_STREAMING
template<int ROWS, int COLS, int TF_param>
void DilationWrap(const uint8_t src[ROWS][COLS], uint8_t dst[ROWS][COLS])
```

The syntax of the pragma as shown in the prior example is:

```
#pragma XMC THROUGHPUT_FACTOR TF_param: 1,2,4
```

Where:

- The `TF_param` must be an `int` type template parameter, as is in the example above.
- It is optional, though recommended, to specify any specific throughput factors that are supported by the function. In the example above, `1,2,4` specifies the supported throughput factors in the pragma, expressed as positive integers, and must include the value 1. If you do not explicitly specify the throughput factors, the `TF_param` is assumed to be valid for any positive throughput factor up to the upper limit of 16 that is supported by Model Composer.

As discussed in Controlling the Throughput of the Implementation, you specify the throughput factor for the model in the Model Composer Hub block. You can specify a throughput factor for the Hub block that divides evenly into one of the `THROUGHPUT_FACTOR` values on the `xmcImportFunction` block.

IMPORTANT! *If the throughput factor of the Hub block does not match, or does not divide evenly into the `THROUGHPUT_FACTOR` specified by the `xmcImportFunction` block, then the throughput is reduced to 1 for the block function.*

Please note the following requirements:

- `THROUGHPUT_FACTOR` pragma must be used on Template functions.
- `THROUGHPUT_FACTOR` pragma must be used with `SUPPORTS_STREAMING` pragma.

- Only one THROUGHPUT_FACTOR pragma can be specified for an xmcImportFunction block.
- The block function will be called with actual arguments that have cyclic ARRAY_RESHAPE directives with factor=TF (see example below). For more information on the ARRAY_RESHAPE pragma, refer to HLS Pragmas in the *Vitis Unified Software Platform Documentation* (UG1416).
- The read accesses from a non-scalar input argument of the function should be compliant with the requirements for streaming, and Vitis HLS should be able to combine groups of TF reads into 1 read of the reshaped array.
- The write accesses into a non-scalar output argument of the function should be compliant with the requirements for streaming, and Vitis HLS should be able to combine groups of TF writes into 1 write of the reshaped array.

The following is an example function specifying both SUPPORTS_STREAMING and THROUGHPUT_FACTOR pragmas:

```
#include <stdint.h>

#pragma XMC THROUGHPUT_FACTOR TF: 1, 2, 4, 8, 16
#pragma XMC SUPPORTS_STREAMING
template<int TF>
void mac(const int32_t In1[240], const int32_t In2[240], const int32_t
In3[240],
         int32_t Out1 [240])
{
    #pragma HLS ARRAY_RESHAPE variable=In1 cyclic factor=TF
    #pragma HLS ARRAY_RESHAPE variable=In2 cyclic factor=TF
    #pragma HLS ARRAY_RESHAPE variable=In3 cyclic factor=TF
    #pragma HLS ARRAY_RESHAPE variable=Out1 cyclic factor=TF

    for (uint32_t k0 = 0; k0 < 240 / TF; ++k0) {
        #pragma HLS pipeline II=1
        int32_t Product_in2m[TF];
        int32_t Sum_in2m[TF];
        int32_t Product_in1m[TF];
        int32_t Sum_outm[TF];
        for (uint32_t k1 = 0; k1 < TF; ++k1) {
            Product_in2m[k1] = In2[(k0 * TF + k1)];
        }
        for (uint32_t k1 = 0; k1 < TF; ++k1) {
            Sum_in2m[k1] = In3[(k0 * TF + k1)];
        }
        for (uint32_t k1 = 0; k1 < TF; ++k1) {
            Product_in1m[k1] = In1[(k0 * TF + k1)];
        }
        for (uint32_t k1 = 0; k1 < TF; ++k1) {
            int32_t Product_in2s;
            int32_t Sum_in2s;
            int32_t Product_in1s;
            int32_t Product_outs;
            int32_t Sum_outs;
            Product_in2s = Product_in2m[k1];
            Sum_in2s = Sum_in2m[k1];
            Product_in1s = Product_in1m[k1];
            Product_outs = Product_in1s * Product_in2s;
            Sum_outs = Product_outs + Sum_in2s;
            Sum_outm[k1] = Sum_outs;
        }
        for (uint32_t k1 = 0; k1 < TF; ++k1) {
            Out1[(k0 * TF + k1)] = Sum_outm[k1];
        }
    }
}
```

Adding Your Library to Library Browser

To use the imported blocks in your library, you can simply open the Simulink model of your library, and copy blocks into a new Model Composer model. However, if you want to see your library listed in the Library Browser, and be able to drag and drop blocks from the library into new models, after running the `xmcImportFunction` command you must prepare your library using the following process.

1. Enable Library Browser parameter.
2. Save the library.
3. Create `slblocks.m` script for the library.
4. Add path to MATLAB, or add library to MATLAB path.
5. Refresh Library Browser.

TIP: *Setting up the library using this process is only required for newly created libraries, rather than existing libraries which have already been setup.*

To enable the library to be available in the Library Browser, you must turn on the `EnableLBRepository` parameter for the library. After importing a block into a new library using `xmcImportFunction`, with the library open, you must use the following command from the MATLAB command line prior to saving your library:

```
set_param(gcs,'EnableLBRepository','on');
```

This parameter identifies the library as belonging to the Library Browser. However, that is just the first step. Save the library by using the **File→Save** command from the main menu, or by clicking on the 💾 button in the toolbar.

Libraries in the Library Browser also require the presence of a script in the same directory as the library model, called `slblocks.m`, that defines the metadata associated with the library. You can create this script by copying an existing script from another library, or copying it from the MATLAB installation, or by editing the following text and saving it as `slblocks.m`:

```
function blkStruct = slblocks
  % This function adds the library to the Library Browser
  % and caches it in the browser repository

  % Specify the name of the library
```

```
    Browser.Library = 'newlib';
    % Specify a name to display in the library Browser
    Browser.Name = 'New Library';

    blkStruct.Browser = Browser;
```

Notice the `Browser.Library` specifies the name of the library model minus the `.slx` file extension, and `Browser.Name` specifies the display name that will appear in Library Browser.

Note: Each library should appear in a separate directory, with the `<library>.slx` file and the `slblocks.m` script for that library.

After creating the library and the `slblocks.m` script, you need to either add the library location to the MATLAB path so that MATLAB can find it, or copy the library to a folder that is already on the MATLAB path. You can type `path` at the MATLAB command prompt to see the current path that MATLAB uses. You can also add the library directory to the MATLAB path using the following commands:

```
addpath ('library_folder')
savepath
```

Where:

- `'library_folder'`
- `savepath` is a string that saves the current search path to `pathdef.m`.

TIP: *To remove a folder from the MATLAB path you can use the following command sequence:*

```
rmpath ('library_folder')
savepath
```

Finally, to view the new library in the Library Browser, from the left side of the Library Browser window right-click and select the **Refresh Library Browser** command. This will load the library into the tool. You should now be able to view the imported blocks and drag and drop them into your models.

Debugging Imported Blocks

Model Composer provides the ability to debug C/C++ code that has been imported as a block using the `xmcImportFunction` command, while simulating the entire design in Simulink®. In this case, the `xmcImportFunctionSettings` command can be used to set the debugging tool.

This feature lets you build the C/C++ function with debug information, and load it into Simulink for simulation. Once loaded, you can step into the C/C++ code of a specific imported block, and debug the function. The debugging environment lets you set debug break points in the C/C++ code, step through, and observe the intermediate results to verify the function in the context of the simulation. Debugging C/C++ code during Simulink simulation provides a natural flow. You can set desired input stimulus in Simulink, and observe the effect stepping through the code.

The debug flow in Model Composer uses the following steps:

1. Specify the debug tool using the `xmcImportFunctionSettings` command.
2. Launch the debugging tool.
3. Add a breakpoint in the imported function.
4. Attach to the MATLAB® process.
5. Start Simulink simulation.
6. Debug the imported function during simulation.

Enable Debug Mode

Debugging the C/C++ function requires the simulation model to be built in the debug mode, instead of being built for release. You will use the `xmcImportFunctionSettings` command to configure the build mode prior to launching simulation. To enable the debug build, use the following command:

```
xmcImportFunctionSettings('build', 'debug')
```

Refer to xmcImportFunctionSettings Command Syntax for more information on the command options. Enabling debug mode in Windows operating system causes Model Composer to return the following messages in the MATLAB® Command Window:

```
Imported C/C++ code will be built with MinGW compiler. You can use gdb to
debug your C/C++ code.
MATLAB process ID is 4656.
You can also get the process ID by typing "feature getpid" in the MATLAB
command window.
```

The information returned above can be used to launch the default debug tool, and connect to the MATLAB process, as described in the following sections.

TIP: *You can restore the release build environment, using the 'release' value of the 'build' option:*

```
xmcImportFunctionSettings('build','release')
```

Launch the Debug Tool

After enabling the debug build mode, the `xmcImportFunctionSettings` command returns a link to the suggested debugging tool.

A third-party debugger is required for debugging with Model Composer. The default debugger is GDB for both the Linux and Windows operating systems.

Setting a Breakpoint for the Imported Function

When the debugger is launched, you can set a breakpoint for the C/C++ imported function in the current model. This will break, or pause the Simulink simulation at the point it enters the imported function. This lets you perform further debugging actions, such as stepping through the function, printing variable values, or listing lines of code. Refer to the documentation for your debugging tool for more information on specific commands, and debugging techniques.

TIP: *The following commands are provided for GDB, as it is the default debugger for Model Composer.*

Setting a breakpoint uses the function name of the imported function:

```
(gdb) break <function_name>
```

Because simulation has not yet started, GDB will respond that no symbol table is loaded, and indicate that you can use the "file" command to specify break points. This simply means that you can also specify breakpoints based on the source file for the imported C/C++ function, and line number, specified as follows:

```
(gdb) break <file name>:<line num>
```

For example: `break func3_d.h:10`

IMPORTANT! *Blocks created from function templates, as described in Defining Blocks Using Function Templates, require the file name and line number to set breakpoints.*

Connecting Debug to the MATLAB Process

With the breakpoint established, you can link GDB to the MATLAB® process by using the GDB attach command, and specifying the process ID (PID) returned by the `xmcImportFunctionSettings` command when you set up the debug build mode, as previously discussed. Use the following command:

```
(gdb) attach <PID>
```

IMPORTANT! *After it is attached to GDB, the MATLAB process will be suspended. You must use the* `continue` *command to have the process resume running after the* `gdb` *prompt is returned:*

```
(gdb) continue
```

At this point you are ready to start the Simulink® simulation, and begin debugging your design.

xmcImportFunctionSettings Command Syntax

The `xmcImportFunctionSettings` command sets options for the import function feature in Model Composer. Options are specified in the form of name/value pairs. The current options include:

- `build`: Specifies the build environment for Model Composer. Supported values include:
  - `release`: The default build mode. Generates the specified output, and lets you perform simulation.
  - `debug`: Provides integration into a third-party compiler to let you step through and observe the imported C/C++ function using standard debug features.
- `compiler`: Specifies the third-party compiler to use for debugging purposes. The supported values are:
  - `default`: Compile with GCC or MinGW compiler for debugging using GDB debugger. Note that GCC or MinGW, and GDB are included with the standard installation of Model Composer.
- `blocks`: Specifies the specific block or blocks that you want to observe during the debugging process. If the option is not specified, all imported function blocks in the current design are included for debugging. You can specify one or more blocks using the following command form:

  ```
  xmcImportFunctionSettings('blocks', {'block1','block2', ...})
  ```

  You can also unset the selection of the blocks using the following command:

  ```
  xmcImportFunctionSettings('blocks', {})
  ```

  You can use the MATLAB `gcb` command to get the block path for a specific block. The blocks must be specified as a list of blocks, even if only one block is specified. For example:

  ```
  xmcImportFunctionSettings('blocks', {'xmc_optical_flow/Lucas-Kanade'})
  ```

TIP: *The settings specified by* `xmcImportFunctionSettings` *remain set until you exit Model Composer, or change the options using* `xmcImportFunctionSettings`*.*

***Note*:** In earlier versions of Model Composer, there existed a set of Computer Vision libraries which are removed from 2020.2 release. You can import these functions into Model Composer as a block using the `xmcImportFunction` feature. To do this, you need to write a wrapper file around the library function. Refer to the Model Composer HLS examples, Sobel Edge detection and Color Detection (from GitHub), to understand how to write a wrapper file and import it as a block into the Model Composer design.

For example: Type, `xmcOpenExample('color_detection')` in the MATLAB command window. You can get the latest version of Vision libraries from this GitHub repository.

Generating Outputs

Introduction

IMPORTANT! *To generate output from the Model Composer HLS model, only HLS library blocks and a limited set of Simulink blocks can be used in the subsystem that is instantiated at the top-level of the design. The Simulink blocks compatible with output generation in Model Composer can be found in the HLS blockset. Refer to Supported Simulink Blocks for a complete list.*

Model Composer automatically compiles designs into low-level representations. However, a Model Composer model requires the addition of the Model Composer Hub block to configure compilation and generate outputs. Model Composer can create three different types of output from the model, as defined by the **Target** setting of the Model Composer Hub block:

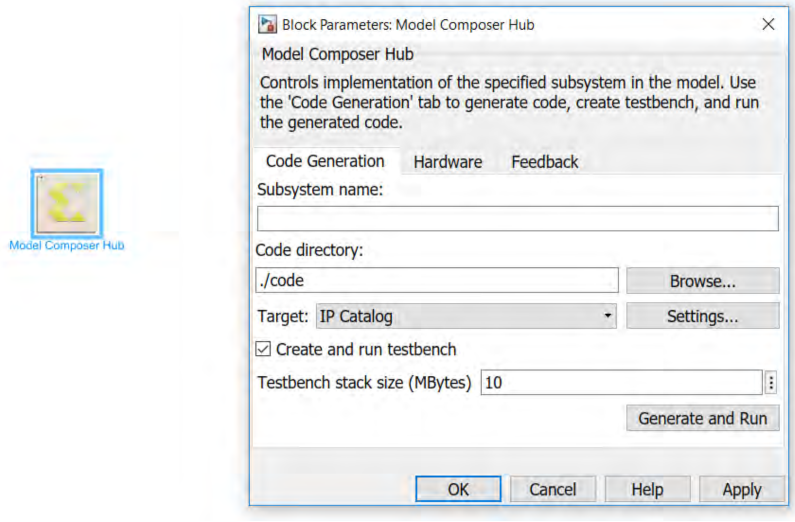
- IP Catalog
- System Generator
- HLS C++ Code

Adding the Model Composer Hub

The Model Composer Hub block is a member of the **Tools** blockset within the HLS library. You can add it to your model just like any other block, by dragging it from the Library Browser onto the canvas of the Simulink® Editor. The Model Composer Hub block is a virtual block in Simulink® terms. It does not provide a physical purpose in the design, but rather provides directives for the compilation and output of the design.

The Model Composer Hub block and the Block Parameters dialog box are shown below.

Figure 189: **Model Composer Hub Block**



Model Composer Hub

Controls implementation of the specified subsystem in the model. Use the 'Code Generation' tab to generate code, create testbench, and run the generated code.

Subsystem name:

Code directory: ./code

Target: IP Catalog

☑ Create and run testbench

Testbench stack size (MBytes) 10

You can see the Block Parameters dialog box of the Model Composer Hub block has three tabs, with the following options:

- **Code Generation** tab
  - **Code directory**: Defines the output folder where the compiled results will be written. The output generated by Model Composer can include a number of folders and files, which will all be written into the specified directory. The folder can be specified as an absolute path (e.g., C:/Data/Code), or a path relative to the current Model Composer model (e.g., ./code).

    IMPORTANT! *On the Windows operating system, you cannot specify a* ***Target Directory*** *name with a space in it.*

  - **Subsystem name**: Specifies the name of the Model Composer subsystem located in the top-level of the model that is required to generate output. Refer to Creating a Top-Level Subsystem Module for more information. The subsystem should also have an Interface Specification block as discussed in Defining the Interface Specification. The subsystem name determines the name of the generated output files (e.g., `<subsystem_name>.cpp`,`<subsystem_name>.h`).

    TIP: *The Interface Spec block is not required to generate output, but it is recommended in order to have full control over the interface specification of the design.*

  - **Target**: Select one of the available output products from Model Composer. These include AI Engine code, packaged IP to add to the Vivado Design Suite IP catalog, HDL blocks for use in RTL level block design, and C++ code for use in Vitis HLS.

    Note: For information about the AI Engines target, refer to Chapter 4: AI Engine Library.

- **Create and run testbench**: When enabled this checkbox causes a simulation testbench to be created and launched for the output code. The simulation compares the "golden" results from Simulink with the results obtained from the newly compiled design. Refer to Simulating and Verifying Your Design for more information.
    - **Testbench stack size**: When **Create and execute testbench** is enabled, this option specifies the size, in Megabytes, of the testbench stack frame used during C simulation (CSIM).

> **TIP:** *Occasionally, the default stack frame size of 10 MB allocated for execution of the testbench may be insufficient to run the test, due to large signals arrays allocated on the stack and deep nesting of sub-systems. Typically when this happens, the test would fail with a segmentation fault and an associated error message. In such cases you may increase the size of the stack frame and re-run the test.*

- **Generate**: Compiles and writes the output from the Model Composer model.

• **Hardware** tab

- **Project device**: Defines the current target part or board platform for the Model Composer model.
- Browse button (**...**): Displays the Device Chooser dialog box. Refer to Device Chooser Dialog Box for more information.
- **FPGA Clock Frequency** (MHz): Specifies the clock frequency in MHz for the Xilinx device. This frequency is passed to the downstream tool flow.
- **Throughput Factor**: Specifies the data throughput requirement for the application, effecting the amount of data resources used in implementing the function in hardware.

• **Feedback** tab

- **Connect**: You can provide feedback on the Vitis Model Composer tool, highlight problems or difficulties, suggest enhancements to the tool, or recommend new blocks for the HLS library.

When you have specified the target directory, the subsystem module, and the export type, you are ready to click the **Generate and Run** button to create the specified output. In addition to producing the C++ code, Model Composer generates the files needed to verify the code, and the necessary directives to synthesize the high-level code into RTL.

> **TIP:** *If you make any changes to the settings of the Model Composer Hub block, you will need to* ***Apply*** *those changes prior to clicking Generate.*

Controlling the Throughput of the Implementation

Introduction

Throughput of a system is one of its most important design criteria. For example, if you are designing a system that processes High Definition video frames (1920x1080) at 30 frames per second, the required throughput of your application would be 62,208,000 pixels per second (1920x1080x30). If you process one pixel per clock (in hardware terminology this is called an initiation interval of one, or II=1), your device needs to be clocked at over 62.2 MHz. If your requirements change, and you need to process a 4K video frame at 60 frames per second, the required throughput of the application would be 497,664,000 pixels per second (3840x2160x60), and your device needs to be clocked at over 497 MHz.

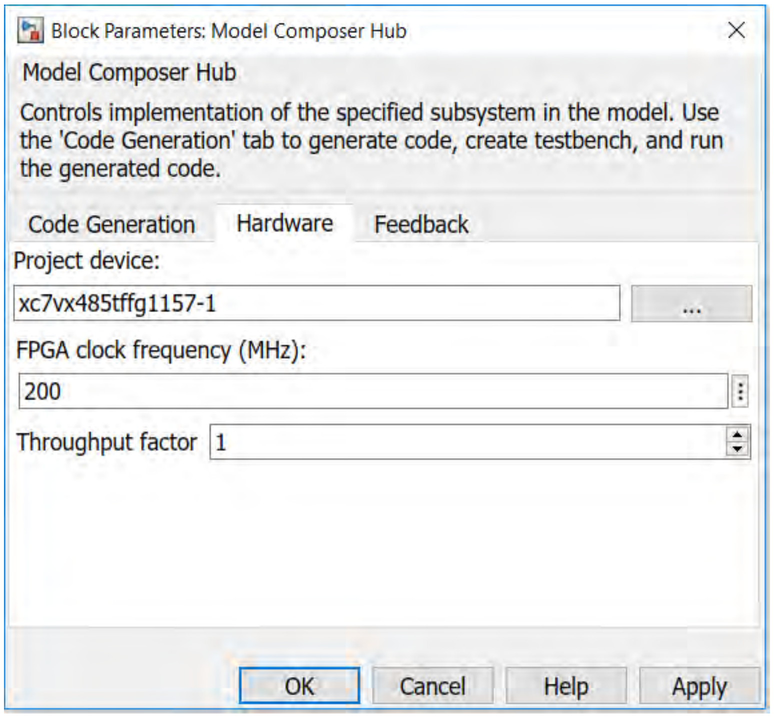
However, in practice you may not be able to achieve an initiation interval of one (II=1), therefore to achieve the desired throughput, you need to clock the device at even higher rates. In other applications, such as wireless communications, the clock frequencies needed to achieve a desired throughput could easily surpass the maximum clock frequency allowed for a device.

If you need to increase the throughput of your design, without increasing the clock frequency that your device is operating at (to operate at a clock frequency below the maximum allowed for a device, or to curtail power consumption), you can take advantage of the programmable logic nature of Xilinx FPGAs, and use parallelization techniques to process more samples per clock. Throughput control in Model Composer allows the user to do exactly that without making structural changes to their design in Simulink.

Setting Throughput Factor from the Hub block

The Model Composer Hub block provides Throughput Factor to control the throughput of the generated code, or hardware design. The Throughput Factor specifies how many sample elements of the inputs are to be processed per clock cycle. By default, the Throughput Factor is set to 1. You can specify this factor by clicking on the **Hardware** tab of the Model Composer Hub block, as shown below:

Figure 190: **Model Composer Hub - Throughput Factor**


Block Parameters: Model Composer Hub

Model Composer Hub

Controls implementation of the specified subsystem in the model. Use the 'Code Generation' tab to generate code, create testbench, and run the generated code.

Project device: xc7vx485tffg1157-1

FPGA clock frequency (MHz): 200

Throughput factor 1

The Throughput Factor must be between 1 and 16. Specifying a value greater than 1 will create parallel logic to process the transactions, using more resources from the device, and increasing the HLS and Vivado Synthesis run time.

Code generation for designs with Throughput Factor > 1 imposes additional restrictions on the design. In case these restrictions are not met, Model Composer will return an error trying to explain the violation.

Restrictions on Using Throughput Control

After you set the value of Throughput Factor, and click the **Generate** button on the Hub, it triggers the compilation of the subsystem. A Throughput Factor of more than 1 can be achieved only if the design complies with the following restrictions:

- The Throughput Factor must be between 1 and 16.
- The subsystem must have at least one non-scalar input port.
- All of the non-scalar ports of the subsystem must use either AXI4-Streaming or FIFO interfaces.
- For any vector signal within the subsystem, but not inside a Window Processing block kernel, the vector length must be a multiple of the Throughput Factor.
- For any matrix signal within the subsystem, but not inside a Window Processing block kernel, the number of columns must be a multiple of the Throughput Factor.

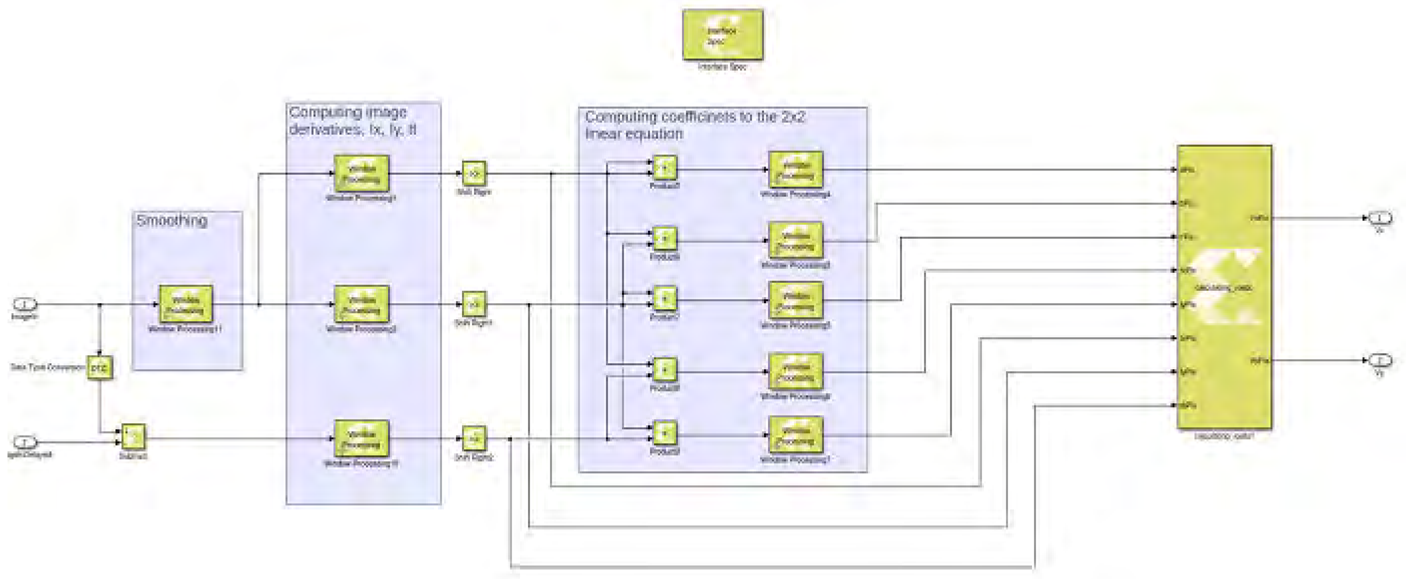
- Except for blocks inside a Window Processing block kernel, the subsystem must not include any of the following blocks:
  - look up
  - matrix multiply, QR inverse
  - transpose, hermitian
  - sum of elements and product of elements with floating point input
  - cumulative sum, reducing min, reducing max
  - if action subsystem
- For blocks created using `xmcImportFunction`, refer to XMC THROUGHPUT_FACTOR.

In summary, if the specified Throughput Factor is >1, and the design complies with all above mentioned restrictions, Model Composer can generate models that process samples concurrently. In cases where the design does not meet these restrictions, Model Composer will not generate an output model.

Understanding Throughput Control Through an Example

The following section demonstrates the benefits of using the Throughput Control feature with the Optical flow example design found in the list of examples for Model Composer HLS library.

Figure 191: **Optical Flow Example**

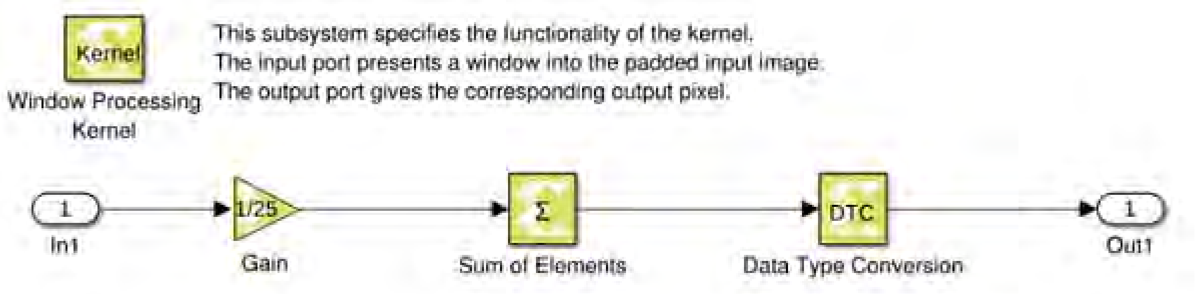


This design uses the following blocks:

- Data Type Conversion, Subtract, Right Shift, Product

- Window processing blocks, with Gain, Sum of Elements, and Data Type Conversion.
- An Import Function block with the `calculating_roots` function.

Figure 192: **Window Processing Kernel**


Kernel
Window Processing Kernel
This subsystem specifies the functionality of the kernel.
The input port presents a window into the padded input image.
The output port gives the corresponding output pixel.
1
In1
1/25
Gain
Σ
Sum of Elements
DTC
Data Type Conversion
1
Out1

All these blocks follow the element-wise application pattern, and comply with the restrictions previously discussed.

IMPORTANT! *Direct use of the Sum of Elements block in subsystems using Throughput Control is restricted. In this example, the 'Sum of Elements' block is used in the Window Processing block but not directly in the top-level subsystem.*

With the default Throughput Factor=1, Model Composer generates the code shown below:

```
void
Lucas_Kanade(hls::stream< uint8_t >& ImageIn, hls::stream< uint8_t >&
    ImageInDelayed, hls::stream< float >& Vx, hls::stream< float >& Vy)
{
    #pragma HLS INTERFACE axis port=ImageIn
    #pragma HLS INTERFACE axis port=ImageInDelayed
    #pragma HLS INTERFACE axis port=Vx
    #pragma HLS INTERFACE axis port=Vy
    #pragma HLS INTERFACE s_axilite port=return
    #pragma HLS dataflow
```

The IP reads its inputs, the image and delayed image, over AXI4-Stream. These streams will use a data width of 8 bits (1 pixel). Similarly pixels of the output image are streamed over an AXI4-Stream interface of data width 8 bits.

If we set TF=4, we get the code shown below.

```
void
Lucas_Kanade(hls::stream< xmc::MultiScalar< uint8_t, 4 > >& ImageIn,
    hls::stream< xmc::MultiScalar< uint8_t, 4 > >& ImageInDelayed,
    hls::stream< xmc::MultiScalar< float, 4 > >& Vx,
    hls::stream< xmc::MultiScalar< float, 4 > >& Vy)
{
    #pragma HLS INTERFACE axis port=ImageIn
    #pragma HLS data_pack variable=ImageIn
    #pragma HLS INTERFACE axis port=ImageInDelayed
    #pragma HLS data_pack variable=ImageInDelayed
    #pragma HLS INTERFACE axis port=Vx
    #pragma HLS data_pack variable=Vx
    #pragma HLS INTERFACE axis port=Vy
    #pragma HLS data_pack variable=Vy
    #pragma HLS INTERFACE s_axilite port=return
    #pragma HLS dataflow
```

This IP receives 4 pixels of the input image, and 4 pixels of the delayed input image, at the same time over AXI4-Stream that have data width of 32 bits. Inside the IP the logic has been duplicated so that 4 pixels are processed in parallel. The IP sends 4 pixels of the output image at a time over an AXI4-Stream, of data width 32 bits. Note that `xmc::MultiScalar<T,N>` is a template `struct` defined in `xmcMultiScalar.h`. It is a `struct` that contains an array of N elements of type T.

The following table represents the Vitis HLS timing and resource estimates for optical flow design.

Table 12: **Optical Flow Design Timing/Resource Utilization Estimates**

	Throughput factor = 1	Throughput factor = 4	Throughput factor = 8
Clock Freq	300 MHz	300 MHz	300 MHz
Latency/II	41848/41834	10483/10469	5358/5344
BRAM_18k (Utilization %)	5	2	4
DSP48E (Utilization %)	2	9	19
FF (Utilization %)	8	30	59
LUT (Utilization %)	14	36	88

The second line in the table shows the initiation interval (II). At clock frequency of 300 MHz and Throughput Factors 4 and 8, the initiation interval of the design is reduced by a factor of approximately 4 and approximately 8 respectively, when compared with the initiation interval for Throughput Factor=1. Note that this comes at the cost of increasing resource utilization when the Throughput Factor increases.

For Throughput factor of one, the II is 41,848. The input to this design is a 200x200 pixel image frame and the value of II here indicates the number of clocks to process the entire frame. As such it takes slightly more than the duration of one clock cycle to process one pixel. As the Throughput Factor increases, the II to process one frame decreases, and the application processes more than one pixel per clock cycle.

Defining the Interface Specification

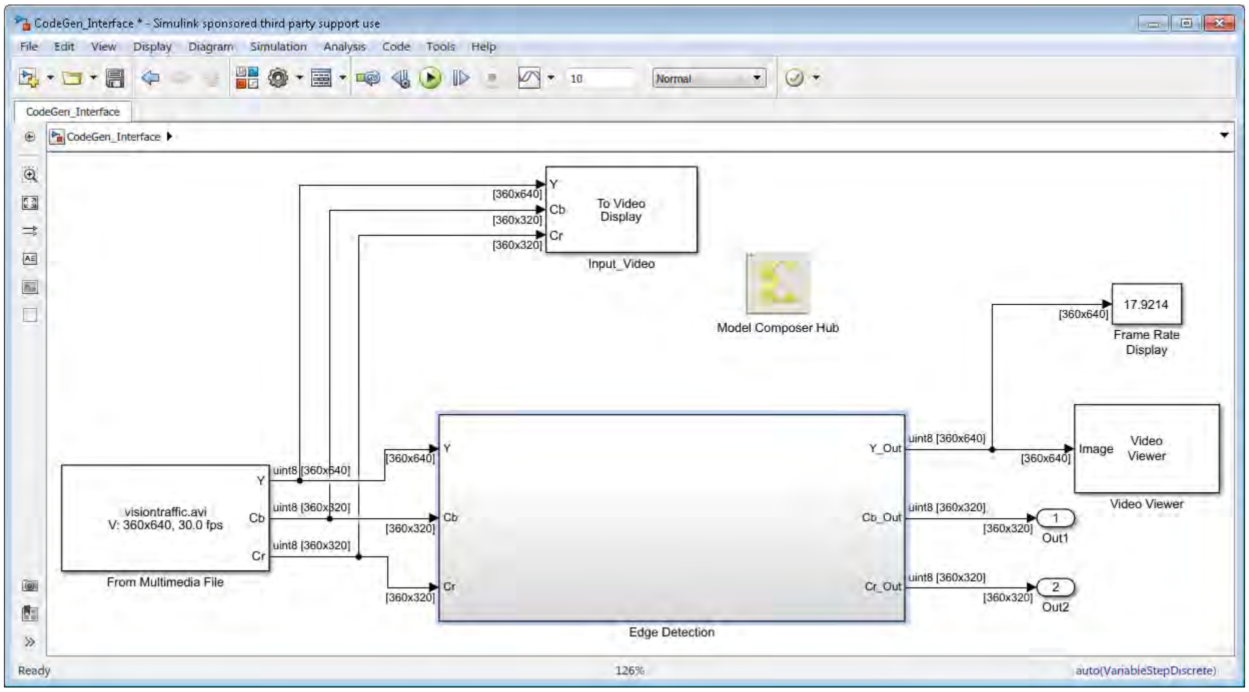
Within the Simulink® environment, the inputs and outputs in your design are defined using "Inport" and "Outport" blocks. However, while moving from the software algorithm to an RTL implementation in hardware, these same input and output ports must be mapped to ports in the design interface, using a specific input-output (I/O) protocol, which typically operates with some real world delay. Part of developing your design is to specify how your design will communicate with other designs or IP in the system. You do this by specifying the interface to your design and choosing among a few standard I/O protocols.

Model Composer requires the use of the Interface Specification (Interface Spec) block to define this I/O protocol.

Interface synthesis is supported only in the top-level subsystem module in the design, which Model Composer generates C++ code for. In the figure below, the Edge Detection module is the top-level subsystem module and the Interface Spec block must be instantiated inside that module.

TIP: *Any Interface Spec blocks instantiated in other subsystems modules, or nested subsystem modules are ignored.*

Figure 193: **Top-Level Subsystem Module**



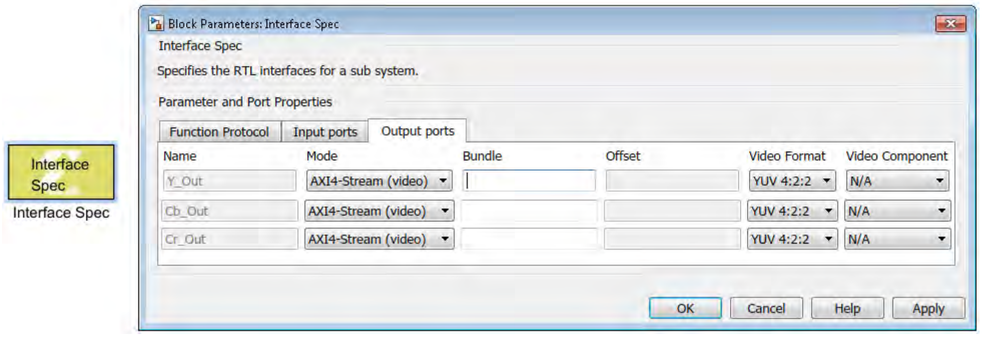
The Interface Spec block lets you control what interfaces should be used for the design. The Interface Spec affects only output code generation; it has no effect on Simulink simulation of your design. If you do not add an Interface Spec block to the subsystem module, Model Composer assigns default interfaces that may not be appropriate for the target platform or device. Therefore, it is recommended that you use the Interface Spec block to define the requirements of your subsystem module. The default function-level protocol is Handshake to specify control signals, and AXI4-Lite Slave for the function return. The default I/O protocol is AXI4-Lite Slave for scalar ports, and AXI4-Stream for non-scalar ports.

The Interface Spec block specifies how RTL ports are created from the function definition during interface synthesis. The ports in the RTL implementation are derived from the following.

- Any function-level protocol that defines control signals for the module.
- Function input and output arguments, and return values.
- Global variables accessed by the function but defined outside its scope.

  Note: If a global variable is accessed, but all read and write operations are local to the subsystem, the resource is created in the design, and does not require the definition of a port.

Figure 194: **Interface Spec Block**



Block Parameters: Interface Spec

Interface Spec

Specifies the RTL interfaces for a sub system.

Parameter and Port Properties

Function Protocol | Input ports | Output ports

Name	Mode	Bundle	Offset	Video Format	Video Component
Y_Out	AXI4-Stream (video)			YUV 4:2:2	N/A
Cb_Out	AXI4-Stream (video)			YUV 4:2:2	N/A
Cr_Out	AXI4-Stream (video)			YUV 4:2:2	N/A

OK | Cancel | Help | Apply

The Interface Spec block consists of 3 tabs defining the following information:

- **Function Protocol**: This is the block-level interface protocol which adds signal ports to the subsystem telling the IP when to start processing data. It is also used by the IP to indicate whether it accepts new data, or whether it has completed an operation, or whether it is idle.
- **Input Ports**: This tab automatically detects the input ports in your subsystem and lets you specify the interface protocol on those ports.
- **Output Ports**: This tab automatically detects the output ports on the subsystem module, and lets you specify the interface protocol.

IMPORTANT! *The Interface Spec block has a current limitation of 8 input ports and 8 output ports on the subsystem module.*

The Interface Specification displays and lets you configure the following features or parameters of the function or I/O port protocol.

Table 13: **Function Protocol Tab**

Attribute	Description
Mode	Specifies a block-level protocol to add control signals to the subsystem module. The supported block-level protocols are: • **AXI4-Lite Slave**: Implements the return port as an AXI4-Lite Slave interface, and adds the block-level control ports defined by the Handshake protocol. This is the default function protocol. • **Handshake**: Defines a set of block-level control ports for the function to `start` processing input, and indicate when the design is `idle`, `done`, and `ready` for new input data. • **No block-level I/O protocol**: No control ports are added to the subsystem.
Bundle	Only valid with the AXI4-Lite Slave mode. Indicates that multiple ports should be grouped into the same interface. The bundle is specified by a `<name>` that cannot contain spaces or special characters.

Table 14: **Input/Output Port Tabs**

Attribute	Description
Name	Displays the port name, which cannot be changed from here.

Table 14: **Input/Output Port Tabs** *(cont'd)*

Attribute	Description
Mode	Specifies the port-level I/O protocols. The supported port-level protocols are: • **Default**: Uses AXI4-Lite Slave interface for scalar ports, or AXI4-Stream interface for non-scalar ports. • **AXI4-Stream**: Implements ports as an AXI4-Stream interface for high-speed streaming data. • **AXI4-Stream (video)**: Implements ports as an AXI4-Stream interface, with the additional assignment of **Video Format** and **Video Component** attributes. • **AXI4-Lite Slave**: Implements the port as part of an AXI4-Lite Slave interface. All input or output ports with the same **Bundle** name are grouped into the same AXI4-Lite Slave interface. • **FIFO**: Implements the port with a standard FIFO interface, combining data input or output with associated active-low FIFO empty and full control signals. ***Note***: The FIFO interface is the most hardware-efficient approach for access to a memory element that is always sequential, that is, no random access is required. To read from non-sequential address locations, use the Block RAM interface. • **Constant**: The data applied to the input port remains stable during the function operation, but is not a constant value that can be optimized. This allows internal optimizations to remove unnecessary registers. • **Valid Port**: Implements a data port with an associated `valid` port to indicate when the data is valid for reading or writing. • **No protocol**: No protocol. Neither the input or output data signals have associated control ports to indicate when data is read or written. • **Block RAM**: Implements array arguments as a standard RAM interface. If you use the generated IP in Vivado IP integrator, the memory interface appears as a single port.
Bundle	Used in conjunction with the AXI4-Stream (video) interfaces that have more than 1 color component. In this case there should be one port for each color component, and the ports should specify the same bundle `<name>` so they will be grouped into the same AXI4 Stream (video) interface. Also valid with the AXI4-Lite Slave mode. This parameter explicitly groups all interface ports with the same bundle `<name>` into the same AXI4-Lite Slave interface.
Offset	Only valid with the AXI4-Lite Slave mode. This parameter specifies an address offset associated with the port in the AXI4-Lite Slave address map. The offset is specified as a non-negative integer, with a default value of 0.
Video Format	Only valid with the AXI4-Stream (video) mode. This parameter specifies the color format for a video stream. Valid formats include: • **YUV 4:2:2**: Video format based on brightness (luminance) and color (chrominance), with reduced color content. • **YUV 4:4:4**: Video format based on brightness (luminance) and color (chrominance), with full color content. • **RGB**: Video format based on separate Red, Blue, and Green color signals. • **Mono**: Specifies an audio format for your video.

Table 14: **Input/Output Port Tabs** *(cont'd)*

Attribute	Description
Video Component	Only valid with the AXI4-Stream (video) mode. This parameter specifies the color component for a video format that uses more than one color component. Valid video components include: • **Y,U, V**: Specifies one component of the YUV video format. • **R, G, B**: Specifies one component of the RGB video format.

The choice of port-level interface protocol should take into account the following considerations:

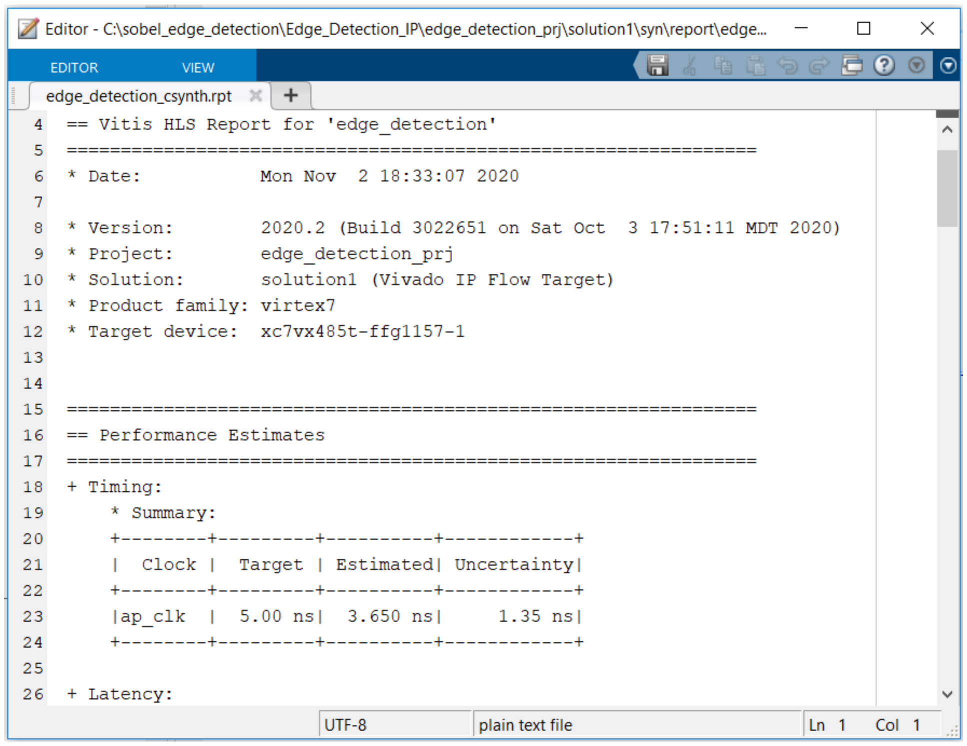
- Scalar ports can be implemented using any of the following protocols: Default, AXI4-Lite Slave, Constant, Valid Port, No protocol.
- Large array or matrix ports should use a streaming protocol such as AXI4-Stream, FIFO, or AXI4-Stream (video).
- Video signals can be transported over an AXI4-Stream (video) interface. In this case you also need to specify the video format YUV 4:2:2, YUV 4:4:4, RGB, or Mono. For video formats that have more than 1 color component, you also need to assign multiple ports to the same signal bundle, and you need to specify which port carries which color component. All of the ports that make up the video signal are implemented by a single AXI4-Stream interface that includes start-of frame and end-of-line sideband signals. For more information refer to *AXI4-Stream Video IP and System Design Guide* (UG934).

Generating Packaged IP for Vivado

Model Composer can automatically generate packaged IP for use in Vivado IP catalog. When Model Composer generates output for the IP catalog, it first writes the C++ code as described in Generating C++ Code, and then it synthesizes RTL code from the C++ code. This process begins when you set the **Target** in the Model Composer Hub block to **IP catalog**, hit **Apply** to confirm any changes, and click **Generate**.

Model Composer displays a transcript window of the process. When the process has concluded, the MATLAB® window displays the Synthesis Report for your review, as shown in the figure below. The Synthesis Report includes details on the estimated performance and resource utilization of the RTL design synthesized by Model Composer. You can review this report to see the estimates and review your model.

Figure 195: **Synthesis Report**


Editor - C:\sobel_edge_detection\Edge_Detection_IP\edge_detection_prj\solution1\syn\report\edge...
EDITOR
VIEW
edge_detection_csynth.rpt
UTF-8
plain text file
Ln 1
Col 1

```
== Vitis HLS Report for 'edge_detection'
================================================================
* Date:           Mon Nov  2 18:33:07 2020

* Version:        2020.2 (Build 3022651 on Sat Oct  3 17:51:11 MDT 2020)
* Project:        edge_detection_prj
* Solution:       solution1 (Vivado IP Flow Target)
* Product family: virtex7
* Target device:  xc7vx485t-ffg1157-1


================================================================
== Performance Estimates
================================================================
+ Timing:
    * Summary:
    +--------+---------+----------+------------+
    |  Clock |  Target | Estimated| Uncertainty|
    +--------+---------+----------+------------+
    |ap_clk  |  5.00 ns|  3.650 ns|     1.35 ns|
    +--------+---------+----------+------------+

+ Latency:
```

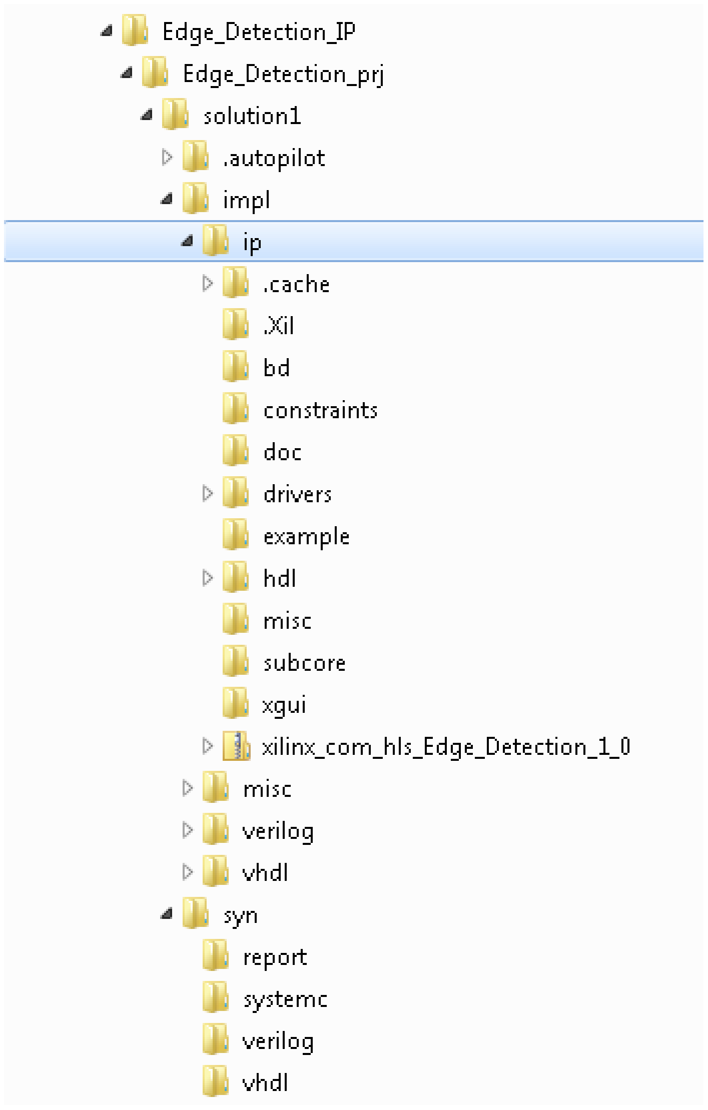
When Model Composer has completed synthesizing the RTL, it reports the message `Exporting RTL as a Vivado IP` to the transcript window, and launches Vivado to create and package the IP for the subsystem design.

Model Composer generates the following outputs from the algorithm:

- SystemC (IEEE 1666-2006, version 2.2)
- VHDL (IEEE 1076-2000)
- Verilog (IEEE 1364-2001)
- Report files created during synthesis, C/RTL co-simulation, and IP packaging.

When Model Composer has completed generating the packaged IP, it can be found in the project directory structure as shown in the following figure. The `Edge_Detection_IP` folder is the **Code Directory** specified by the Model Composer Hub. The `Edge_Detection_prj` folder is a project created by the `run_hls.tcl` script. The `solution1` folder is a Vitis HLS solution. For more information refer to the *Vitis High-Level Synthesis User Guide* (UG1399). The `syn` and `impl` folders store the results of synthesis and implementation. The `ip` folder contains the packaged IP to add to the Vivado Design Suite IP catalog.

Figure 196: **Packaged IP Folder**



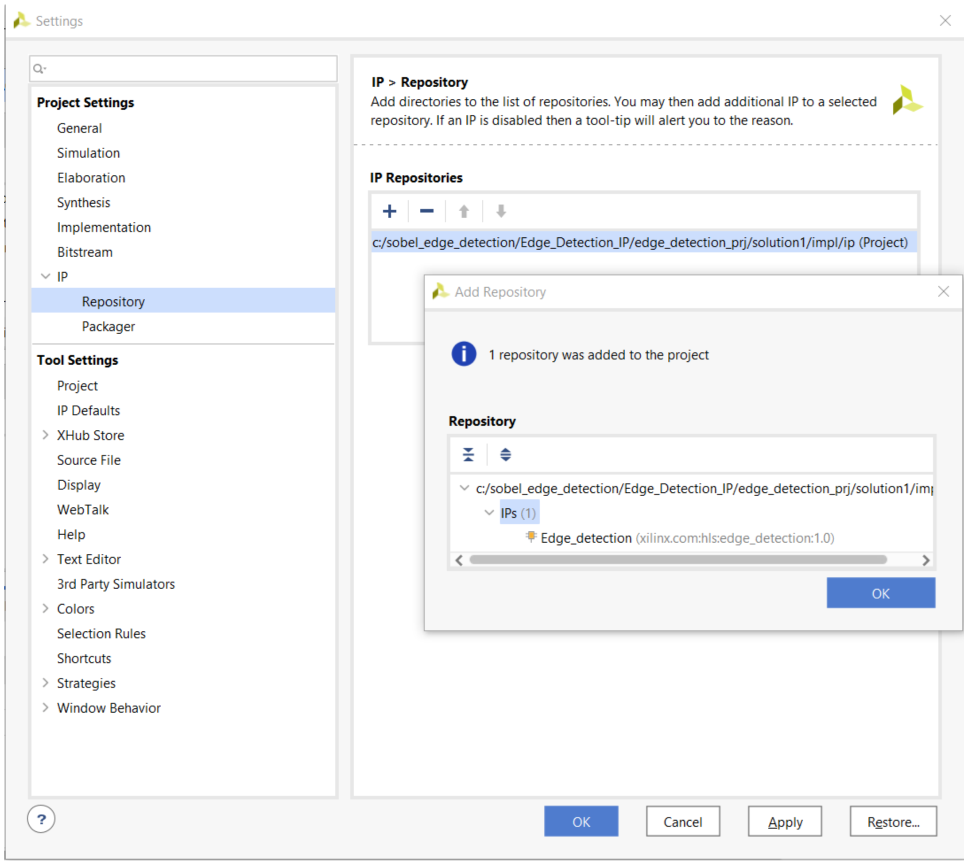
```
Edge_Detection_IP
  Edge_Detection_prj
    solution1
      .autopilot
      impl
        ip
          .cache
          .Xil
          bd
          constraints
          doc
          drivers
          example
          hdl
          misc
          subcore
          xgui
          xilinx_com_hls_Edge_Detection_1_0
        misc
        verilog
        vhdl
      syn
        report
        systemc
        verilog
        vhdl
```

After Model Composer has generated the packaged IP, the `.zip` file archive in the `<project_name>/<solution_name>/impl/ip` folder can be imported into the Vivado IP catalog, and used in any Vivado Design Suite design, either as RTL IP, or in the IP integrator.

For Model Composer models that specify AXI4-Lite Slave interfaces through the Interface Specification block, as discussed in Defining the Interface Specification, a set of software driver files is also created by Vitis HLS during the IP packaging process. These C driver files can be included in an SDK C project and used to access the AXI4-Lite Slave slave port. The software driver files are written to directory `<project_name>/<solution_name>/impl/ip/drivers` and are included in the packaged IP.

To add the IP into the Vivado IP catalog, from within the Vivado GUI, select the **Tools→Settings** command to open the Settings dialog box. Select the **IP→Repository** command, and add the Vitis HLS packaged IP as shown in the following figure.

Figure 197: **Setting the IP Repository**



Add directories to the list of repositories. You may then add additional IP to a selected repository. If an IP is disabled then a tool-tip will alert you to the reason.

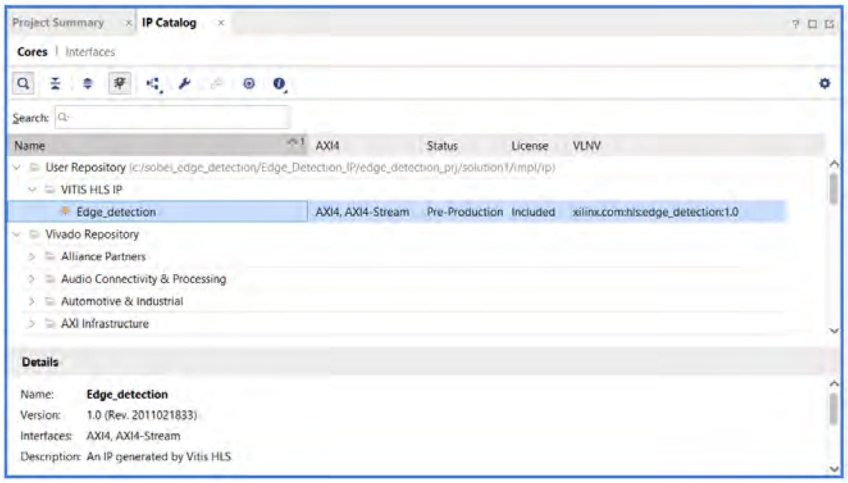
c:/sobel_edge_detection/Edge_Detection_IP/edge_detection_prj/solution1/impl/ip (Project)

1 repository was added to the project

c:/sobel_edge_detection/Edge_Detection_IP/edge_detection_prj/solution1/imp
IPs (1)
Edge_detection (xilinx.com:hls:edge_detection:1.0)

After adding the path to the repository, the IP is added to the IP catalog as shown in the following figure. You can now use the IP in standard RTL designs, or in Vivado IP integrator block designs. For more information on working with IP and adding to the IP repository refer to the *Vivado Design Suite User Guide: Designing with IP* (UG896).

Figure 198: **IP Catalog**



Project Summary | IP Catalog

Cores | Interfaces

Search:

Name	AXI4	Status	License	VLNV
User Repository (c:/sobel_edge_detection/Edge_Detection_IP/edge_detection_prj/solution1/impl/ip)				
VITIS HLS IP				
Edge_detection	AXI4, AXI4-Stream	Pre-Production	Included	xilinx.com:hls:edge_detection:1.0
Vivado Repository				
Alliance Partners				
Audio Connectivity & Processing				
Automotive & Industrial				
AXI Infrastructure				

Details

Name: **Edge_detection**
Version: 1.0 (Rev. 2011021833)
Interfaces: AXI4, AXI4-Stream
Description: An IP generated by Vitis HLS

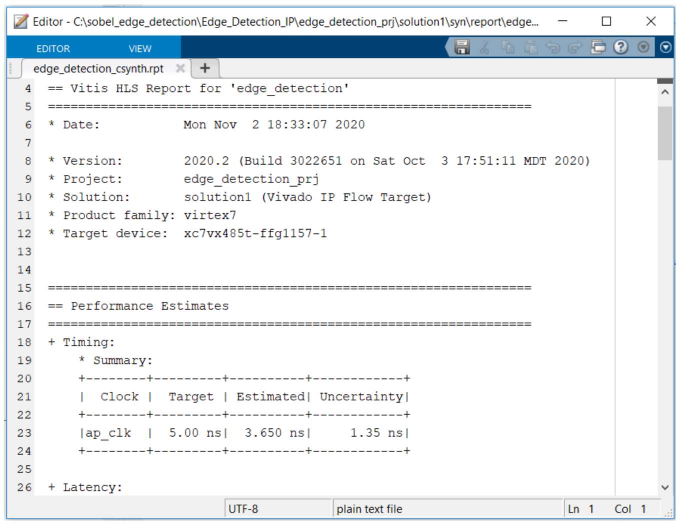
IMPORTANT! *If you see the repository added to the IP catalog, but do not see the Vitis HLS packaged IP, it is possible the target part for the current project is not compatible with the device used when generating the Model Composer output. You can fix this by changing the part in the current project to the device specified by the Model Composer model.*

Generating Model Composer HDL IP

Model Composer can automatically generate an RTL which can be imported and used along side other blocks in the HDL library. When Model Composer generates HDL output, it first writes the C++ code as described in Generating C++ Code, and then it synthesizes RTL code from the C++ code. This process begins when you set the **Target** in the Model Composer Hub block to **System Generator**. This command creates an IP package for HDL library.

Model Composer displays a transcript window of the process. When the process has concluded, the MATLAB window displays the Synthesis Report for your review, as shown in the figure below. The Synthesis Report includes details on the estimated performance and resource utilization of the RTL design synthesized by Model Composer. You can review this report to see the estimates and review your model.

Figure 199: **Synthesis Report**



Editor - C:\sobel_edge_detection\Edge_Detection_IP\edge_detection_prj\solution1\syn\report\edge...

edge_detection_csynth.rpt

```
== Vitis HLS Report for 'edge_detection'
================================================================
* Date:           Mon Nov  2 18:33:07 2020

* Version:        2020.2 (Build 3022651 on Sat Oct  3 17:51:11 MDT 2020)
* Project:        edge_detection_prj
* Solution:       solution1 (Vivado IP Flow Target)
* Product family: virtex7
* Target device:  xc7vx485t-ffg1157-1


================================================================
== Performance Estimates
================================================================
+ Timing:
    * Summary:
    +--------+---------+----------+------------+
    |  Clock |  Target | Estimated| Uncertainty|
    +--------+---------+----------+------------+
    |ap_clk  |  5.00 ns|  3.650 ns|     1.35 ns|
    +--------+---------+----------+------------+

+ Latency:
```

UTF-8 | plain text file | Ln 1 Col 1

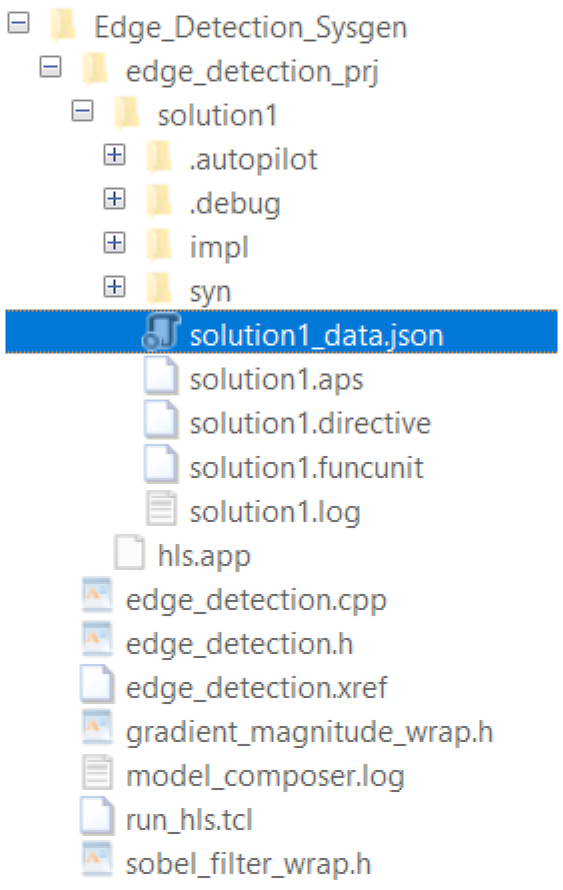
When Model Composer has completed synthesizing the RTL, it reports the message `Exporting RTL as an IP for System Generator for DSP` to the transcript window. This process is handled by a Tcl script, `run_hls.tcl`, that Model Composer writes to export the HDL IP.

Model Composer generates the following outputs from the algorithm:

- SystemC (IEEE 1666-2006, version 2.2)
- VHDL (IEEE 1076-2000)
- Verilog (IEEE 1364-2001)
- Report files created during synthesis, C/RTL co-simulation, and IP packaging.

When Model Composer has generated the HDL IP, you can find it in the project directory structure as shown in the following figure. The `Edge_Detection_Sysgen` folder is the **Code Directory** specified by the Model Composer Hub. The `Edge_Detection_prj` folder is a project created by the `run_hls.tcl` script. The `solution1` folder is a Vitis HLS solution. For more information refer to the *Vitis High-Level Synthesis User Guide* (UG1399). The `Solution1.json` file contains the information needed to use the subsystem IP in Model Composer HDL design..

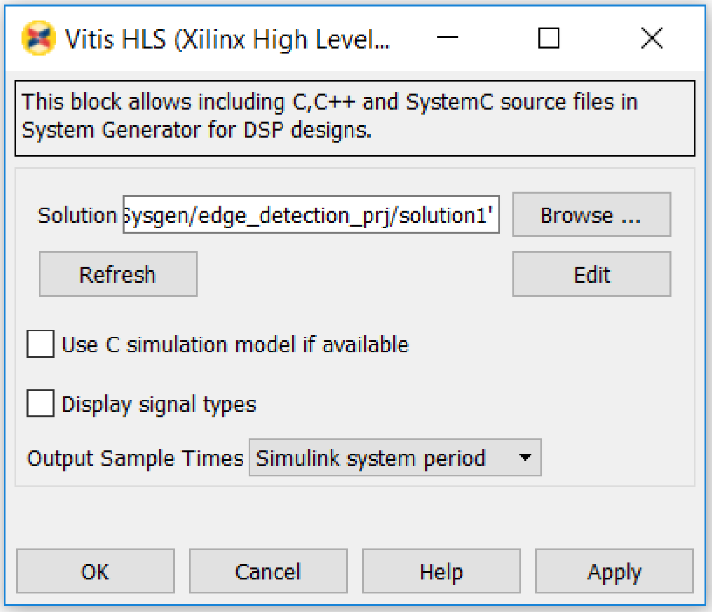
Figure 200: **System Generator Output**


```
Edge_Detection_Sysgen
  edge_detection_prj
    solution1
      .autopilot
      .debug
      impl
      syn
      solution1_data.json
      solution1.aps
      solution1.directive
      solution1.funcunit
      solution1.log
    hls.app
  edge_detection.cpp
  edge_detection.h
  edge_detection.xref
  gradient_magnitude_wrap.h
  model_composer.log
  run_hls.tcl
  sobel_filter_wrap.h
```

You can import a Model Composer generated HDL IP into a Model Composer HDL model using the following steps:

1. From within an open Model Compose HDL model, right-click in the canvas of the Simulink Editor and select the **Xilinx BlockAdd** command. This opens a menu of HDL Library blocks that can be added to your model.
2. Scroll down the list in dialog box, or type "HLS" in the **Add Block** search field to locate the Vitis HLS block and add it to your model.
3. Double-click on the newly added block to open the **Vitis HLS Block Parameter** dialog box as shown below.

Figure 201: **Vitis HLS Block**



This block allows including C,C++ and SystemC source files in System Generator for DSP designs.

Solution: ysgen/edge_detection_prj/solution1'

[ ] Use C simulation model if available

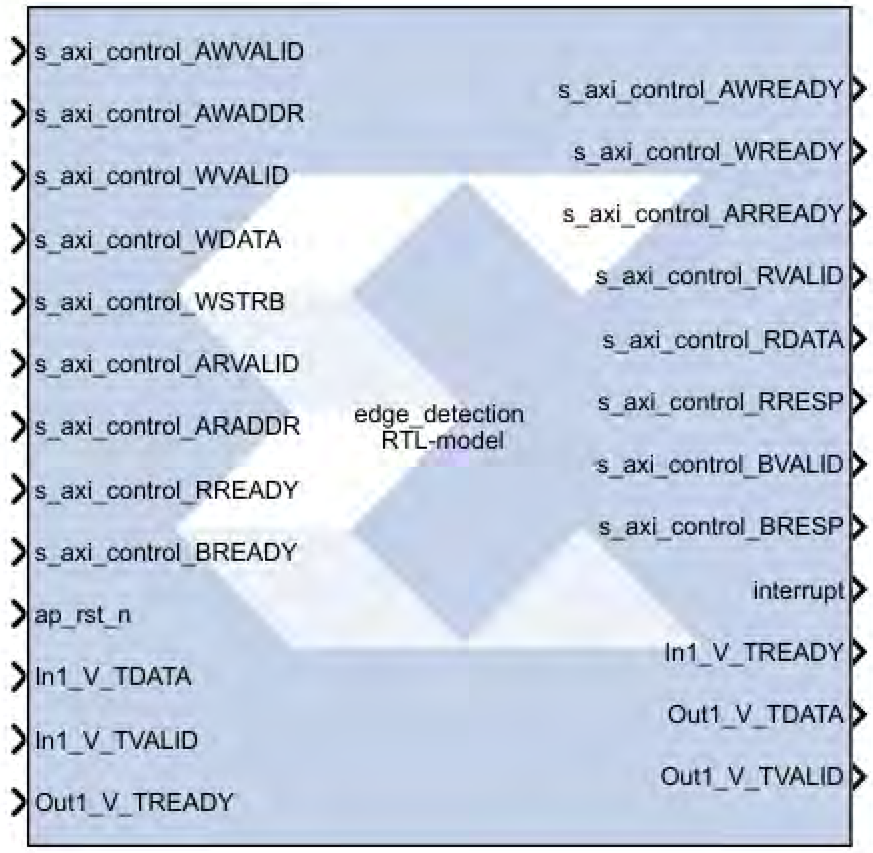
[ ] Display signal types

Output Sample Times: Simulink system period

4. **Browse** to the solution directory of the Vitis HLS project where the Model Composer output was generated. In the example above, browse to the `Lucas_Kanade_prj/solution1` folder and select **OK**.

The Vitis HLS template block is converted to the Edge Detection IP in the Model Composer HDL model. You may need to drag the corners of the IP block to expand it as needed for your model. The block is initially sized to match the Vitis HLS template. The following figure shows the HDL IP generated from the Model Composer HLS model.

Figure 202: **Vitis HLS Block**



If any of the function arguments on the Model Composer subsystem module are transformed by Vitis HLS into a composite port, the `signal type` information for that port cannot be determined and included in the HDL IP block. Any design that uses the reshape, mapping, or data packing optimization on ports must have the port type information manually specified in Model Composer HDL model for these composite ports. You should know how the composite ports were originally created and then use `slice` and `reinterpretation` blocks in the HDL model to connect the Vitis HLS block to other blocks in the system.

For example, if three 8-bit in-out ports R, G and B are packed into a 24-bit input port (`RGB_in`) and a 24-bit output port (`RGB_out`) ports. After the IP block has been included in Model Composer HDL:

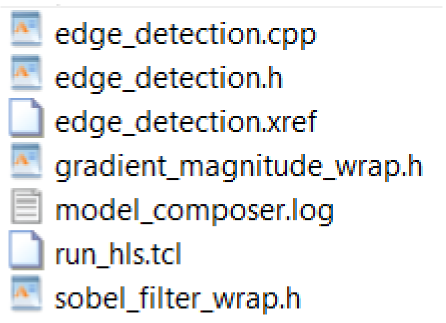
- The 24-bit input port (`RGB_in`) would need to be driven by an HDL block that correctly groups three 8-bit input signals (`R_in`, `G_in` and `B_in`) into a 24-bit input bus.
- The 24-bit output bus (`RGB_out`) would need to be correctly split into three 8-bit signals (`R_out`, `G_out`, and `B_out`).

TIP: *See the Chapter 2: HDL Library documentation for details on using the slice and reinterpretation blocks to connect to composite type ports.*

Generating C++ Code

The following figure shows the HLS C++ Code output by Model Composer from the **Generate** command. The C++ code is output either as an intermediate step when generating a packaged IP for System Generator output, or as a specified output to let you optimize the C++ code using directives or pragmas in Vitis HLS.

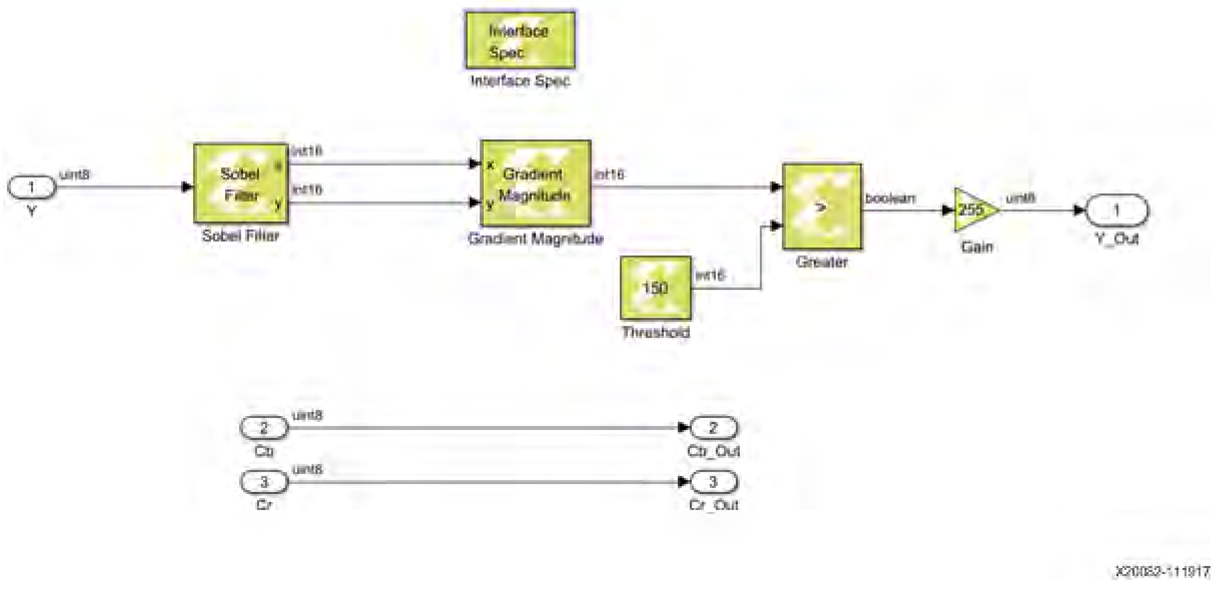
Figure 203: **C++ Output Files**



edge_detection.cpp
edge_detection.h
edge_detection.xref
gradient_magnitude_wrap.h
model_composer.log
run_hls.tcl
sobel_filter_wrap.h

The files generated by Model Composer reflect the contents and hierarchy of the subsystem that was compiled. In this case, the subsystem is the Edge Detection function described in the Model Composer section of the *Vitis Model Composer Tutorial* (UG1498). The following figure shows the contents of the Edge Detection subsystem.

Figure 204: **Edge Detection Subsystem**



The `Edge_Detection.cpp` file specifies the following include files, which incorporate the code generated for the various Model Composer blocks used in the subsystem:

```
#include "Edge_Detection.h"
#include "GradMagnitude.h"
#include "SobelFilter.h"
```

The following shows the generated code for the Edge Detection subsystem. Notice the pragmas added to the function to specify the function protocol and the I/O port protocols for the function signature and return value. The pragmas help direct the solution synthesized by Vitis HLS, and result in higher performance in the implemented RTL.

```
Edge_Detection(hls::stream< ap_axiu<16, 1, 1, 1> >& Y,
    hls::stream< ap_axiu<16, 1, 1, 1> >& Y_Out)
{
    #pragma HLS INTERFACE s_axilite port=return
    #pragma HLS INTERFACE axis bundle=image_out port=Y_Out
    #pragma HLS INTERFACE axis bundle=input_vid port=Y
    #pragma HLS dataflow
    uint8_t core_Y[360][640];
    #pragma HLS stream variable=core_Y dim=2 depth=1
    uint8_t core_Cb[360][320];
    #pragma HLS stream variable=core_Cb dim=2 depth=1
    uint8_t core_Cr[360][320];
    #pragma HLS stream variable=core_Cr dim=2 depth=1
    uint8_t core_Y_Out[360][640];
    #pragma HLS stream variable=core_Y_Out dim=2 depth=1
    uint8_t core_Cb_Out[360][320];
    #pragma HLS stream variable=core_Cb_Out dim=2 depth=1
    uint8_t core_Cr_Out[360][320];
    #pragma HLS stream variable=core_Cr_Out dim=2 depth=1
    fourier::AxiVideoStreamAdapter< uint8_t >::readStreamVf0(Y,
        reinterpret_cast< uint8_t* >(core_Y), reinterpret_cast< uint8_t* >(
        core_Cb), reinterpret_cast< uint8_t* >(core_Cr), 360, 640);
    Edge_Detection_core(core_Y, core_Cb, core_Cr, core_Y_Out, core_Cb_Out,
        core_Cr_Out);
    fourier::AxiVideoStreamAdapter< uint8_t >::writeStreamVf0(Y_Out,
        reinterpret_cast< uint8_t* >(core_Y_Out), reinterpret_cast<
uint8_t* >(
        core_Cb_Out), reinterpret_cast< uint8_t* >(core_Cr_Out), 360, 640);
}
```

Finally, notice the `run_hls.tcl` file that is generated in the output folder. This is a Tcl script that can be used to run Vitis HLS on the generated output files to create a project and solution, synthesize the RTL from the C++ code, and export the design to the Model Composer HDL model. Each Vitis HLS project holds one set of C/C++ code and can contain multiple solutions. Each solution can have different constraints and optimization directives. For more information refer to the *Vitis High-Level Synthesis User Guide* (UG1399).

You can run the `run_hls.tcl` script from a Vitis HLS command prompt as follows:

1. Open the Vitis HLS Command Prompt:
    - On Windows, use **Start → All Programs → Xilinx Design Tools → Vitis HLS 2020.2 → Vitis HLS → Vitis HLS 2020.2 Command Prompt**.

- On Linux, open a new shell and source the `<install_dir>/Vitis_HLS/<version>/settings64.sh` script to configure the shell.

2. From the command prompt, change the directory to the parent folder of the **Code Directory** specified on the Model Composer Hub dialog box when you generated the output, as discussed at Adding the Model Composer Hub. For example:

   ```
   cd C:/Data
   ```

3. From the command prompt, launch the `run_hls.tcl` script that can be found in **C**:

   ```
   vitis_hls -f ./code/run_hls.tcl
   ```

Vitis HLS launches to synthesize the RTL from the C++ code, generating a Vitis HLS project, and solution in the process. You can open the Vitis HLS project by going to the **Code directory** and entering the following name with the project name:

```
vitis_hls -p ./Edge_Detection_proj
```

This will open the Vitis HLS project in the GUI Mode.

Model Composer Log File

To help you diagnose issues related to code generation, Model Composer generates a log file, `model_composer.log`, that is written to the **Code Directory** specified on the Model Composer Hub block.

By default, Model Composer generates code for the model using a streaming micro-architecture in which blocks run concurrently via task pipelining, or dataflow. However, in some cases this streaming micro-architecture is not achievable, because the model includes an `xmcImportFunction` block that does not support streaming for example. In this case, Model Composer generates code in which the blocks operate sequentially. The Model Composer log file includes information to help you identify when this condition occurs, and what some possible causes might be.

The log file also contains information related to which blocks are used in the Model Composer model.

If **Target** on the Model Composer Hub block is set to HLS C++ code, and **Create and run testbench** is selected, the `model_composer.log` file will contain the output from running the C++ verification.

If **Target** is set to **IP Catalog** or **System Generator**, information related to running Vitis HLS is provided. In these cases, more detailed information can be found in the `vitis_hls.log` file which can also be found in the `Target` directory.

Simulating and Verifying Your Design

Introduction

Verification in Model Composer can be separated into two distinct processes:

- Verification of the algorithm in Simulink® to verify the functional correctness of the design.
- Verification of the Model Composer model, to confirm the equivalence of the simulation results in Simulink and the C++ and RTL outputs.

In high-level synthesis, running the compiled C program is referred to as C simulation. Executing the C++ algorithm generated by Model Composer simulates the function and verifies that the output from the code matches the output from the Simulink simulation to validate that the algorithm is functionally correct.

In C/RTL co-simulation, Vitis HLS uses the C test bench to simulate the C function prior to synthesis and to verify the RTL output. The verification process consists of three phases:

1. The C simulation is executed and the inputs to the top-level subsystem are saved as "input vectors".
2. The input vectors are used in behavioral simulation of the RTL code for the top-level subsystem created by Vitis HLS. The outputs from the RTL are saved as "output vectors".
3. The output vectors are applied to the C test bench as output from the top-level subsystem, to verify the results of the C-simulation match.

Vitis HLS uses this return value for both C simulation and C/RTL co-simulation to determine if the results are correct.

Simulating in Simulink

Simulink can interactively simulate your model, and view the results on scopes and graphical displays. The Simulink model defines what data to input at the start of simulation, and defines what data to capture during simulation.

When defining the model, you define the input and output signals for the model. Input signals load data into the model for simulation, while output signals allow you to record simulation results. The kind of data you want to load impacts your choice of signal loading techniques. You can define input data as constant values, use source blocks, such as the Sine Wave block, import data from a spreadsheet, or use the output of a previous simulation. Refer to Prepare Model Inputs and Outputs in the Simulink documentation for more information on preparing for simulation in Simulink.

After the simulation is completed, you can analyze any logged data with MATLAB scripts and visualization tools like the Simulation Data Inspector within the MathWorks environment.

When you enable the **Create and execute testbench** checkbox, as discussed in Adding the Model Composer Hub, Model Composer performs two tasks:

1. Automatically logs the test data, or stimulus, at the input of your design, and the simulation results as test vectors for later use as "golden" data for comparison during C/C++ and RTL co-simulation. This file is named `signals.stim`, and is added to the specified **Target Directory** when generating output.
2. Executes the generated test bench and verifies equivalence of the code using C-simulation and C/RTL co-simulation. This process is compute intensive, and can take considerable time.

IMPORTANT! *If the model simulation time in Simulink is long, with significant amounts of data processed, the test bench execution will take an even longer time. Model Composer will generate an error if the simulation time becomes infinite.*

Managing the HLS Block Cache

Model Composer creates a cache entry when you simulate a model with an imported block created using `xmcImportFunction`. When you simulate the model again, unless you make any changes to the function, or supporting source files, Model Composer will use the cached entry for the block to initiate the simulation faster. You can manage the simulation cache in Model Composer using the following command from the MATLAB command prompt:

```
>> xmcSimCache
```

The usage for this command is as follows.

Table 15: **xmcSimCache Command Usage**

Command	Description
`xmcSimCache('enable')`	Enable the simulation cache. This is enabled by default.
`xmcSimCache('disable')`	Disable the simulation cache.
`xmcSimCache('isEnabled')`	Returns the state of the simulation cache.
`xmcSimCache('setLocation', <dir>)`	Specify a directory to use for the simulation cache. For example: `xmcSimCache('setLocation', 'C:/temp')`
`xmcSimCache('setDefaultLocation')`	Restore the simulation cache to its default location.
`xmcSimCache('getLocation')`	Return the current location of the simulation cache.
`xmcSimCache('clear')`	Clear the entire simulation cache.

Table 15: **xmcSimCache Command Usage** *(cont'd)*

Command	Description
`xmcSimCache('clear', 'release', <version>)`	Clear simulation cache entries for the specified release version. For example: `xmcSimCache('clear', 'release', '2017.4')`
`xmcSimCache('clear', 'days', <number>)`	Clear simulation cache entries older than or equal to the specified number of days. For example: `xmcSimCache('clear', 'days', 30)`
`xmcSimCache('clear', 'id', <vals>)`	Clear simulation cache entries by the cache ID. For example: `xmcSimCache('clear', 'id', {'12345678', 'abcdefgh'})`

Verifying the C++ Code

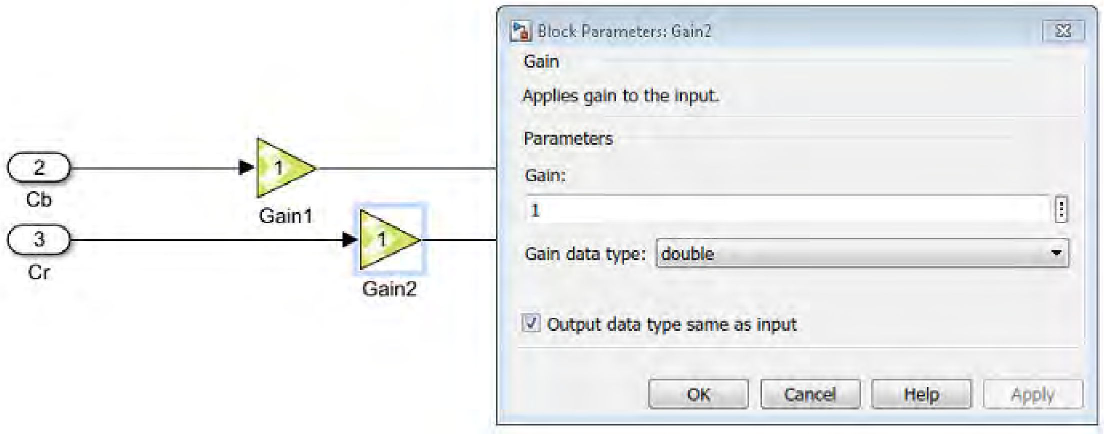
When generating the output using the Model Composer Hub block, you also have the ability to **Create and run testbench**. When selecting this option, you are enabling the Model Composer verification flow. This causes Model Composer to generate a few more output files, including a makefile, the test bench, `tb.cpp`, and `signals.stim` as previously discussed. The purpose of the test bench is to apply input stimuli, generated during Simulink simulation, to the top-level function of the generated C++ or RTL code and compare that function's output against the output samples captured in the `signals.stim` file. Depending on the output generated, the verification flow runs simulation on the C++ or RTL outputs generated by Model Composer and looks for the same result as generated by Simulink.

When the **Export type** on the Model Composer Hub block is HLS C++ code, the verification flow is as follows:

- The model is simulated in Simulink and the input and outputs are logged into the `signals.stim` binary file.
- Model Composer generates the C++ code and a test bench, `tb.cpp`, which contains a `main()` function.
- Model Composer launches simulation.
- It verifies that the output from the generated C++ code matches the output logged from the Simulink simulation, `signals.stim`.
- In case of a mismatch, the mismatched output signal name is reported, as well as the actual and expected values.
- The result is a Pass/Fail returned by Model Composer.

IMPORTANT! *During simulation by Model Composer, you may receive the following error message:* `Failed to find a XilinxLibrary block connected to input port.` *This error simply means that there is an input in the subsystem that does not connect to a block from the HLS Library block set. This may be due to the presence of signals that you are simply passing through the subsystem for signal grouping, or improved readability. The recommended fix for this conditions is to connect the input to a Gain block from the Xilinx Toolbox->**HLS→Math Functions→Math Operations** block library, with the default value of 1, and **Output data type same as input** checked, as shown in the following figure.*

Figure 205: **Adding Gain to Unconnected Inputs**



Verifying the C/RTL Code

When the **Target** specified on the Model Composer Hub dialog box is either IP catalog or System Generator, and the verification flow is enabled, Model Composer uses C/RTL co-simulation to verify the RTL output. Again, the objective is to verify that the results of the RTL simulation match the results of the Simulink simulation. In this case, the verification flow is as follows:

- The Model Composer model is simulated in Simulink to capture the test vectors in `signals.stim`.
- Model Composer generates the C/C++ code and the C test bench, `tb.cpp`.
- Model Composer runs the C-synthesis and generates the RTL output.
- Model Composer runs the C/RTL co-simulation. This step ensures the following:
  - That the C++ code generated by Model Composer is correct by comparing with the Simulink simulation, `signals.stim`.
  - That the RTL code generated by Model Composer is correct by comparing the output stimulus from RTL with C/C++ output.

TIP: *After verification, Model Composer exports the RTL as an IP for Model Composer HDL model, or packages the IP for use in Vivado.*

- The result is a separate Pass/Fail returned by Model Composer for both the C-simulation and the C/RTL co-simulation. If the C-simulation fails, the process stops before the C/RTL simulation is run.

Select Target Device or Board

Device Chooser Dialog Box

Choose the default part or platform board to use for the current project. The device resources that the design is synthesized against, and placed onto are determined by selecting the target part or board.

- **Parts tab:** Lists the available target parts for the current project.
- **Boards tab:** Lists the available target boards for the current project.

IMPORTANT! *The devices and boards that are available are determined at the time the Vitis Model Composer tool is installed. You can also add uninstalled devices or boards. To learn more refer to Xilinx® Answer Record 60112.*

The selection of target part or boards can be limited or filtered by specifying search patterns in one or more of the available search fields at the top of the Device Chooser dialog box.

For parts:

- **Category:** Choose devices according to Military, Automobile, or Commercial grade products.
- **Family:** Filter devices according to the available device families (such as Virtex, Kintex, or UltraScale).
- **Package:** Specify the type of package the device will have.
- **Speed:** Filter the device by a specific speed grade.
- **Temperature:** Filter the device by a specific temperature.

For boards:

- **Vendor:** Choose devices according to the available vendors of platform boards.
- **Name:** Filter devices according to the available display names.
- **Board Rev:** Specify the revision level of the board.

The target parts or boards that match the specified filters and/or search string appear in a table of results in the lower portion of the dialog box.

Choose a target part or board for the design, and click **OK**.

Chapter 4

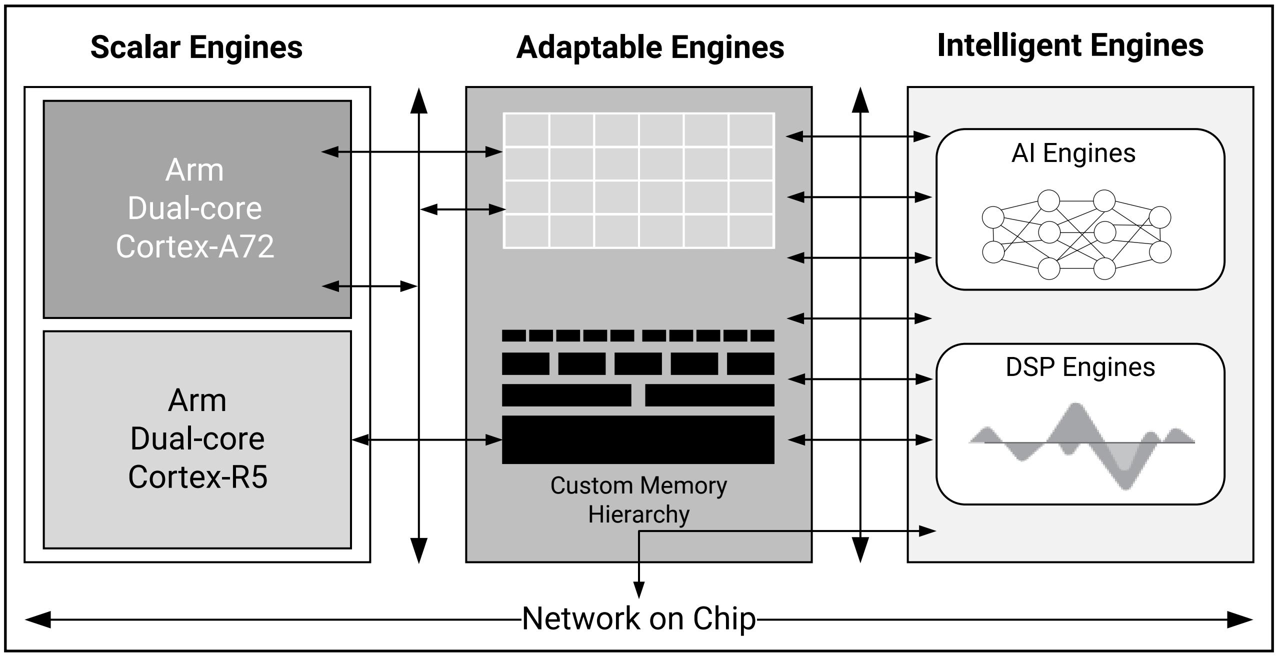
AI Engine Library

Introduction

Versal™ devices are the industry's first adaptive compute acceleration platform (ACAP), combining adaptable processing and acceleration engines with programmable logic and configurable connectivity to enable customized, heterogeneous hardware solutions for a wide variety of applications in Data Center, Automotive, 5G Wireless, Wired, and Defense. Versal ACAPs provide transformational features like an integrated silicon host interconnect shell and Intelligent Engines (AI and DSP), Adaptable Engines, and Scalar Engines, providing superior performance/watt over conventional FPGAs, CPUs, and GPUs.

Versal devices are built from a library of building blocks dedicated to processing, compute, acceleration, and connectivity. The following figure shows a top-level view of the Versal ACAP, that contains three major architectural areas: the Scalar Engines that include the Arm® processing system, the Adaptable Engines that include the programmable logic, and the Intelligent Engines that include the AI Engines and DSP Engines. The AI Engine along with Adaptable Engines (programmable logic) and Scalar Engines (processor subsystem) form a tightly integrated heterogeneous compute platform.

Figure 206: **Xilinx Versal ACAP Overview**



AI Engines

An AI Engine is an array of very-long instruction word (VLIW) processors with single instruction multiple data (SIMD) vector units that are highly optimized for compute-intensive applications, specifically digital signal processing (DSP), 5G wireless applications, and artificial intelligence (AI) technology such as machine learning (ML). They provide up to five times higher compute density for vector-based algorithms.

AI Engines provide multiple levels of parallelism including instruction-level and data-level parallelism:

- Instruction-level parallelism includes two scalar instructions, two vector reads, a single vector write, and a single vector instruction executed—in total, a six-way VLIW instruction per clock cycle.
- Data-level parallelism is achieved via vector-level operations where multiple sets of data can be operated on a per-clock-cycle basis.

Each AI Engine contains both a vector and scalar processor, dedicated program memory, local 32 KB data memory, and access to local memory in any of three neighboring directions (north, south, east, or west). It also has access to DMA engines and AXI4 interconnect switches to communicate via streams to other AI Engines or to the programmable logic (PL) or the DMA.

Adaptable and Scalar Engines

Adaptable Engines are a combination of programmable logic blocks and memory, architected for flexible custom-compute and data movement. Scalar Engines, including Arm Cortex®-A72 and Cortex®-R5F processors, allow for complex control processing tasks.

Refer to the *Versal ACAP AI Engine Architecture Manual* (AM009) for specific details on the AI Engine array and interfaces.

Refer to the *Versal Architecture and Product Data Sheet: Overview* (DS950) for specific details on Compute and Acceleration Engines.

AI Engine Kernels

An AI Engine kernel is a C/C++ program written using AI Engine vector data types and specialized intrinsic calls that target the VLIW vector processor. They are computational functions running on an AI Engine. These kernels form the fundamental building blocks of an AI Engine program which consists of dataflow graph specification. The AI Engine kernel code is compiled using the aiecompiler included in the Vitis™ software platform core development kit. The aiecompiler compiles the kernels to produce an ELF file that runs on the AI Engine processors.

AI Engine Graphs

An AI Engine program consists of a dataflow graph specification written in C++. This specification can be compiled and executed using the aiecompiler. A static dataflow (SDF) graph application consists of nodes and edges where nodes represent compute kernel functions and edges represent data connections. Kernels in the application can be compiled to run on the AI Engine or in the PL region of the device.

Model Composer for AI Engine Development

Model Composer enables the rapid simulation, exploration, and code generation of algorithms targeted for AI Engines from within the Simulink® environment. You can achieve this by importing AI Engines kernels and data-flow graphs into Model Composer as blocks and controlling the behavior of the kernels and graphs by configuring the block GUI parameters. Simulation results can be visualized by seamlessly connecting Simulink source and sink blocks with the Model Composer AI Engines blocks. Furthermore, the simulation results can be sent to the MATLAB® workspace for further analysis.

Refer to Creating an AI Engine Design using Model Composer for more information on importing AI Engine kernels and graphs as blocks.

Model Composer provides a set of AI Engine library blocks for use within the Simulink environment. These include:

- Blocks to import kernels and graphs which can be targeted to the AI Engine portion of Versal devices.
- Block to import HLS kernels which can be targeted to the PL portion of Versal devices.
- Blocks that support connection between the AI Engine and the Xilinx HDL blockset.
- Configurable AI Engine functions such as FIR filters.

***Note*:** For more information on specific blocks refer to Model Composer AI Engine Block library.

Connecting HLS kernel blocks, HDL library blocks, and AI Engine blocks, allows modeling and simulation of a heterogeneous platform which can be targeted to both programmable logic and AI Engines in Versal™ ACAP devices.

In addition to simulation, you can also use the Model Composer Hub to generate dataflow graphs. For more details on the Model Composer Hub block, specific to AI Engine code generation, refer to Code Generation.

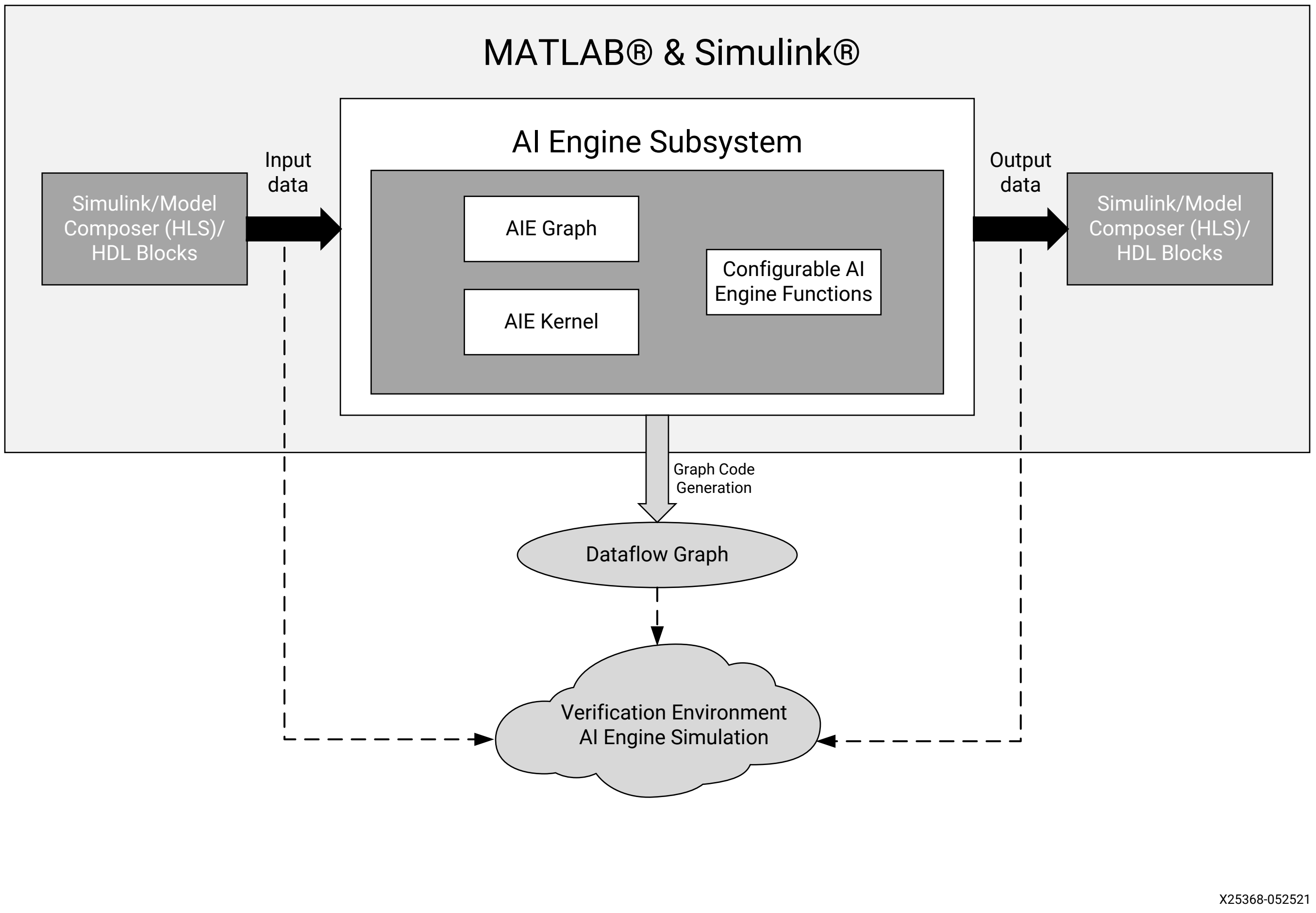
Model composer allows you to verify the generated dataflow graph code using the AI Engine simulator. Based on verification requirements, you can choose to verify your algorithm from the Model Composer Hub block.

The simulation results are compared against the reference design in the Simulink environment.

***Note*:** Refer to Simulation and Code Generation for more details on simulator options and verification.

A typical AI Engine design flow is shown in the following diagram.

Figure 207: **Typical AI Engine Design Flow**


MATLAB® & Simulink®
AI Engine Subsystem
Simulink/Model Composer (HLS)/ HDL Blocks
Input data
AIE Graph
AIE Kernel
Configurable AI Engine Functions
Output data
Simulink/Model Composer (HLS)/ HDL Blocks
Graph Code Generation
Dataflow Graph
Verification Environment AI Engine Simulation
X25368-052521

To learn more about the AI Engine flow in Model Composer, refer to the *Vitis Model Composer Tutorial* (UG1498).

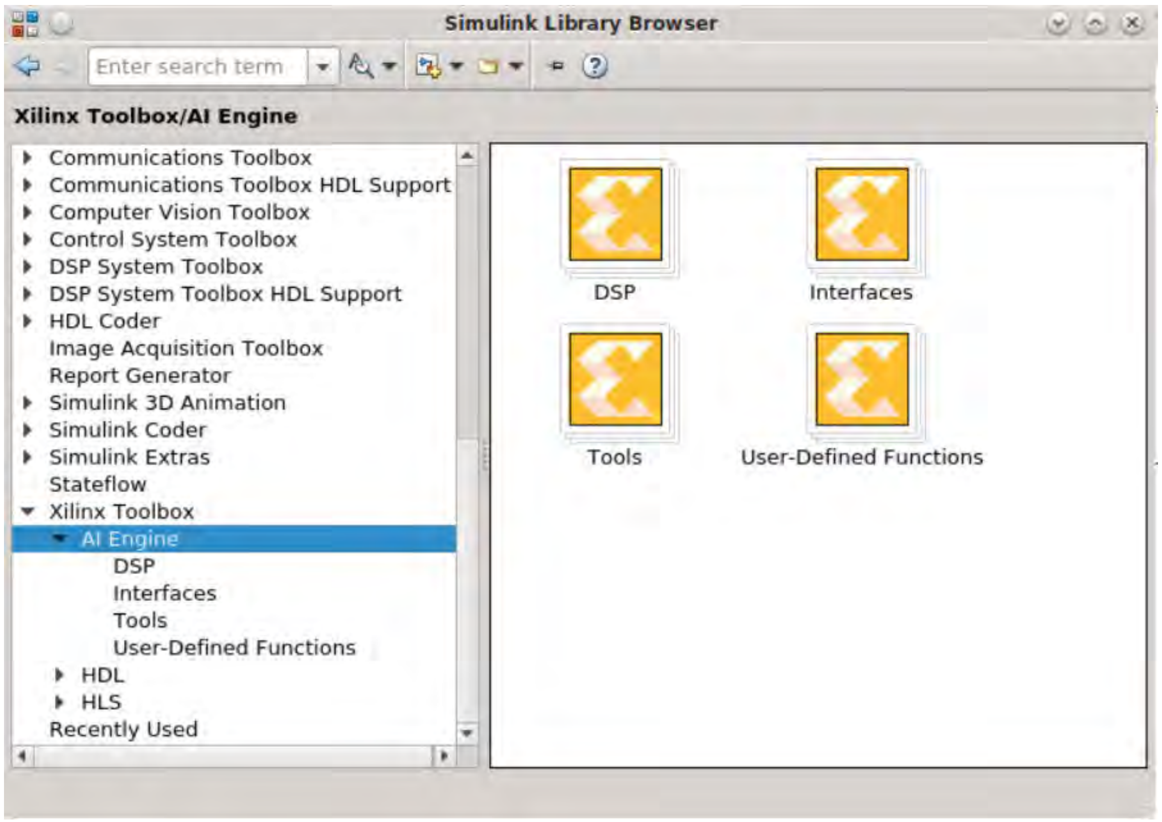
The remainder of this section discusses the following topics:

- Creating an AI Engine Design using Model Composer
- Simulation and Code Generation
- Verification of AI Engine Code

Creating an AI Engine Design using Model Composer

As previously discussed, AI Engine kernels are functions that form the fundamental building blocks of the data-flow graph. Model Composer supports generating the AI Engine data-flow graph by importing the AI Engine kernel or sub-graph. The AI Engine library is available under Xilinx tool box in the Simulink library browser set as shown in the following figure.

Figure 208: **Simulink Library Browser**



Preparing the Kernels

Kernels are declared as C/C++ functions that return void and can use special data types for arguments which are discussed in Data Accessing Mechanisms. The kernels should be defined each in their own source file. This organization is recommended for reusability and faster compilation. Furthermore, the kernel source files should include all relevant header files to allow for independent compilation.

IMPORTANT! *It is assumed that your kernel code is already developed. If you try to import bad kernel code, for example one that causes memory corruption, you may see undesirable outcomes including MATLAB crashing.*

The topics in this section introduce some basics of AI Engine programming which are necessary to understand the Model Composer AI Engine design flow. Refer to *Versal ACAP AI Engine Programming Environment User Guide* (UG1076) for a high-level overview of the kernel and AI Engine programming models. Some insight is also provided on data access APIs to help you understand the configuration parameters of the AI Engine kernel blocks available in the library.

Data Accessing Mechanisms

An AI Engine kernel can either consume or produce blocks of data, or, it can access and produce data streams in a sample-by-sample fashion. The data access APIs for both cases are described in the following sections.

Window-Based Access

From the kernel perspective, incoming blocks of data is called an input window. Input windows are defined by the type of data contained within that window. The following example shows a declaration of an input window carrying complex integers where the real and imaginary parts are both 16 bits wide.

```
input_window_cint16 myInputWindow;
```

From the kernel perspective, outgoing blocks of data is called an output window. Again, these are defined by type. The following example shows a declaration of an output window carrying 32-bit integers.

```
output_window_int32 myOutputWindow;
```

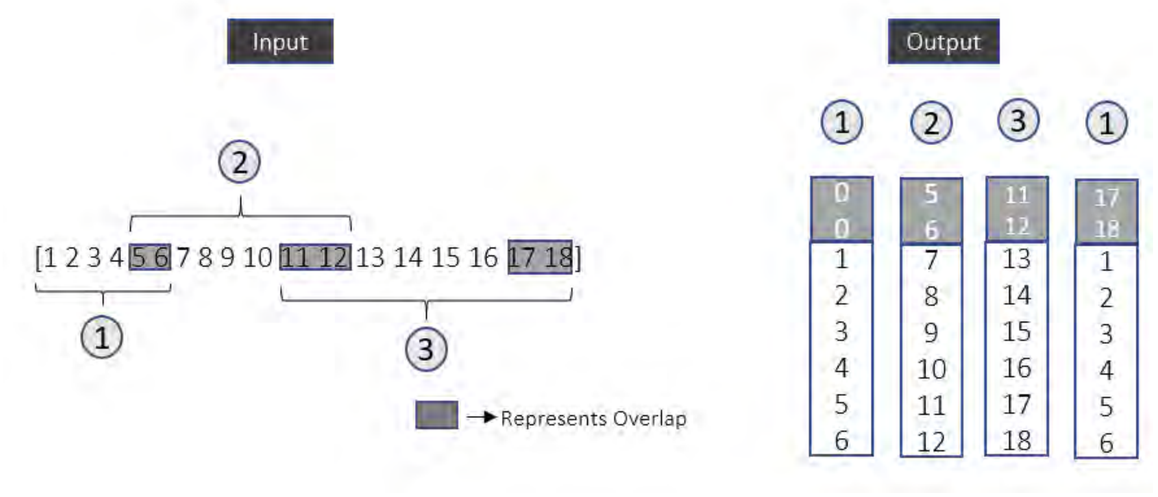
A kernel reads from its input windows and writes to its output windows. By default, the synchronization required to wait for an input window of data or provide an empty output window is performed before entering the kernel. There is no synchronization required within the kernel to read or write the individual elements of data. In other words, the kernel will not execute unless there is a full window available.

In some situations, if you are not consuming a windows worth of data on every invocation of a kernel, or if you are not producing a windows worth of data on every invocation, you can control the buffer synchronization by configuring the kernel port to be `async` in the Kernel GUI block parameters.

It is also possible to have overlap from one block of input to the next. This in general is required for certain algorithms such as filters. This overlap is referred to as 'Window Margin'. If a Window margin is specified, the kernel has access to a total number of samples equal to $\text{window\_size} + \text{margin\_size}$.

The behavior of the window margin can be demonstrated using the following example.

Figure 209: **Simulation**



Here, input is a vector of size `18` and this is fed to the kernel block which is configured to have a window size of `6` with and a window margin of `2`, as shown in the previous figure. The kernel should have access to a total of `8` samples at every invocation. During the first simulation cycle, two `0`'s are prepended to the first `6` new values from the input data. For the subsequent simulation cycles, the kernel receives `8` values which includes `6` new values and `2` values from the previous cycle."

Stream-Based Access

Kernels can access data streams in a sample-by-sample fashion using data access APIs. With a stream-based access model, kernels receive an input stream or an output stream of typed data as an argument. Each access to these streams is synchronized (i.e., reads stall if the data is not available in the stream and writes stall if the stream is unable to accept new data). There is also a direct stream communication channel between one AI Engine and the physically adjacent AI Engine, called a cascade.

The following example shows a declaration of input and output streams of type `cint16`.

```
input_stream_cint16 * myInputStream;
output_stream_cint16 * myOutputStream;
```

Run-Time Parameter Specification

It is possible to modify the behavior of the AI Engine program or data-flow graph based on a dynamic condition or event using the run-time parameter. The modification could be in the data being processed, for example a modified mode of operation or a new coefficient table, or it could be in the control flow of the graph such as conditional execution or dynamically reconfiguring a graph. Either the kernels or the graphs can be defined to execute with parameters.

If an integer scalar value appears in the formal arguments of a kernel function, then that parameter becomes a run-time parameter. Run-time parameters are processed as ports alongside those created by streams and windows. Both scalar and array values can be passed as run-time parameters.

Consider the following example where a kernel function is defined with run-time parameters. Here, `select` is a scalar RTP port and `coefficients` is a vector RTP with 32 integers.

```
#ifndef RTP_KERNEL_H
#define RTP_KERNEL_H
void simple_param(input_window_cint16 * in,
                  output_window_cint16 *outw,
                  int select,
                  const int32 (&coefficients)[32]);
#endif
```

Two types of RTPs are supported:

- **Synchronous Parameters (or triggering parameters):** The kernel does not execute until the run-time parameter is written by a controlling processor. Upon a write, the kernel executes once, reading the new updated value. After completion, it is blocked from executing again until the parameter is updated. This allows for a different type of execution model from the normal streaming model, and can be useful for certain updating operations where blocking synchronization is important.
- **Asynchronous Parameters:** These parameters can be changed any time by either a controlling processor such as Arm® , or another AI Engine. They are read each time a kernel is invoked without any specific synchronization. These types of parameters can be used, for example, to pass new filter coefficients to a filter kernel that changes infrequently.

Importing AI Engine Code as a Block

Model Composer allows you import the AI Engine kernel from the Model Composer AI Engine library. This allows you to create a block with an interface that has input and output ports equivalent to the arguments of an AI Engine kernel function, and also gives the flexibility to configure the kernel parameters using the block GUI. If you have a data-flow graph instead of an AI Engine kernel function, then Model Composer also allows you to import it into the Simulink environment from where it is possible to seamlessly connect the AI Engine graph block with AI Engine kernel block to build a complete system. If you have a kernel function targeted at implementation in adaptable engines (programmable logic), then Model Composer provides an HLS Kernel block. Coding guidelines for importing HLS kernels are described in Importing HLS Kernels.

In summary, the entry point for using an AI Engine block set can be a kernel or a data-flow graph for which Model Composer generates a block with interfaces that match the function arguments of a kernel or a graph.

Variable-Size Signals

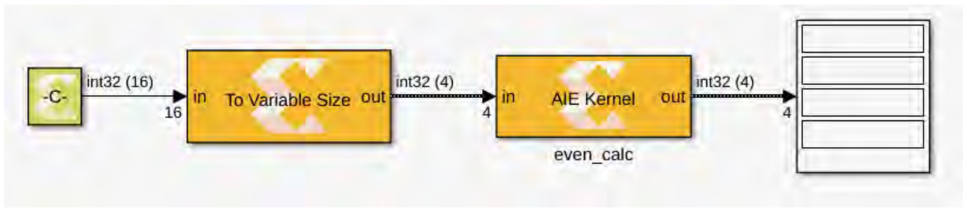
In Simulink, a signal whose size (the number of elements in a dimension) can change during Simulink simulation is called a variable-size signal. To understand the importance of variable-size signaling in the context of modeling an AI Engine design in Model Composer, consider a kernel that outputs even numbers from a set of input numbers.

```
void even_calc(input_window_int32 * in, output_stream_int32 * out) {
    int32 val,temp;
    for (unsigned i=0; i<4; i++) {
        window_readincr(in,val);
        if(val % 2 == 0) {
            int32 temp = val;
            writeincr(out,temp);
        }
    }
}
```

Here, the input is a window of type `int32` and the output is a stream of similar type. If you try to model this in Simulink, you will observe that the number of data samples it produces may vary at each invocation.

Assume the input window size is `4` and the input given to the kernel is `[1 -2 3 -4 5 -6 7 8 9 10 12 14 5 7 9 13]`.

Figure 210: **Variable-Size Signals**



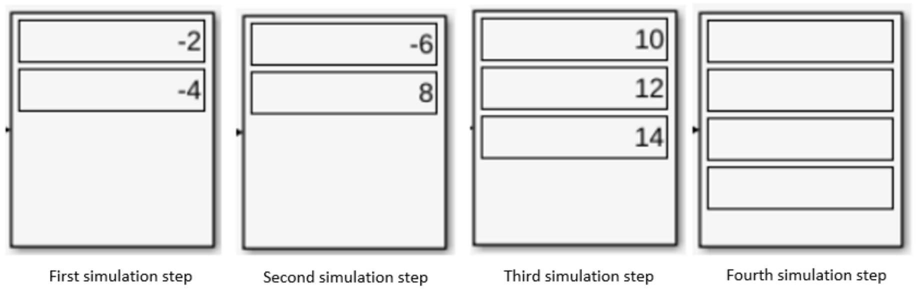
To understand why the size of the output can vary dynamically, run the simulations in steps.

During the first simulation step, the AI Engine kernel consumes four values (i.e., `1,-2,3,-4`) and outputs two values (`[-2,-4]`), which are the even numbers in the set `[1,-2,3,-4]`. In the second simulation step, the kernel consumes the next set of window inputs (i.e, `[5,-6,7 ,8]`) the output is `[-6,8]`. For the third set of inputs `[9,10,12,14]` , the output at the third simulation step is `[10,12,14]`. Similarly for the final set of inputs `[5,7,9,13]`, the output is empty because there are no even numbers to produce from the fourth input window.

In summary, the output size at the:

- First simulation step is `2`.
- Second simulation step is `2`.
- Third simulation step is `3`.
- Fourth simulation step is `0`.

Figure 211: **Display Block - Simulation Steps**



So, the kernel produces data samples of different sizes at every invocation and sometimes it does not produce any output. To model this behavior in Model Composer, you can use variable-size signals in Simulink. However, during a simulation, the number of dimensions cannot change. A variable-size signal always has an associated maximum signal size. In this case, it is `4`.

The variable-size signal in Simulink is represented with a thicker signal line unlike the normal signal as highlighted in the previous figure. You can learn more about variable size signals at the following links:

- MATLAB
- GitHub

Importing AI Engine Kernels

Model Composer supports importing C/C++ kernels functions. Function must have `void` as the return type. It also supports importing class kernels as well as templatized kernels. Model Composer provides two AI Engine library blocks to import kernel functions of different types (class-based and non-class-based kernels):

- AIE Kernel

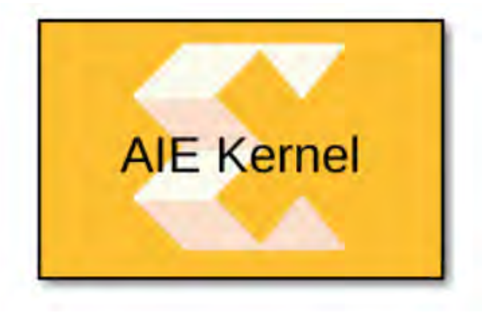
- AIE Class Kernel

as described in the following sections.

Non-Class-Based Kernels

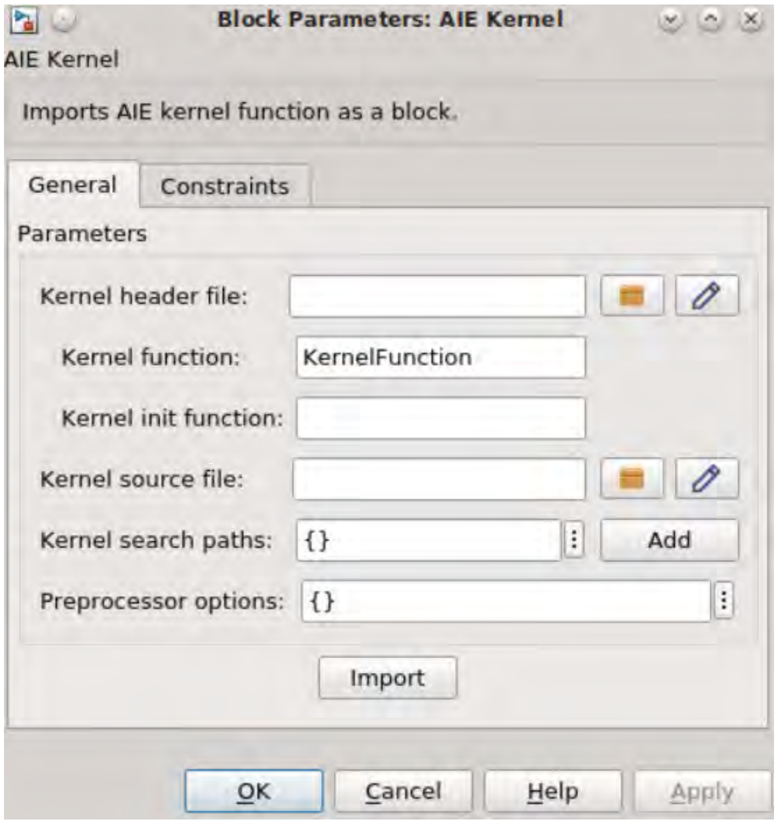
To Import a non-class-based AI Engine kernel as a block into Model Composer, you need to use an AI Engine kernel block from the AI Engine Library as shown.

Figure 212: **AI Engine Kernel**



Double-clicking the block symbol displays the parameters of the AI Engine kernel block as shown in the following figure.

Figure 213: **Block Parameters: AI Engine Kernel**


Block Parameters: AIE Kernel

AIE Kernel

Imports AIE kernel function as a block.

General | Constraints

Parameters

Kernel header file:

Kernel function: KernelFunction

Kernel init function:

Kernel source file:

Kernel search paths: {}

Preprocessor options: {}

The block mask parameters need to be updated in order to import the kernel function as a block. The following table provides details on the parameters and description for each parameter.

Table 16: **AI Engine Kernel: Master Parameters**

Parameter Name	Parameter Type	Criticality	Description
Kernel header file	String	Mandatory	Name of the header file that contains the kernel function declaration. The string could be just the file name, a relative path to the file, or an absolute path of the file. Use the **Browse** button to navigate to the file.
Kernel function	String	Mandatory	Name of the kernel function for which the block is to be created. This function should be declared in the kernel header file.
Kernel init function	String	Optional	Name of the initialization function used by the kernel function.
Kernel source file	String	Mandatory	Name of the source file that contains the kernel function definition. The string could be the file name, a relative path to the file or an absolute path of the file.
Kernel search paths	Vector of Strings	Optional	If the kernel header file or the kernel source file are not found using the value provided through Kernel header file or Kernel source file fields, respectively, then the paths provided through Kernel search paths are used to find the files. This parameter allows the use of environment variables while specifying paths for the kernel header file and the kernel source file. The environment variable can be used in either `${ENV}` or `$ENV` format.

Table 16: **AI Engine Kernel: Master Parameters** *(cont'd)*

Parameter Name	Parameter Type	Criticality	Description
Preprocessor options		Optional	Optional preprocessor arguments for downstream compilation with specific preprocessor options. The following two preprocessor option formats are accepted and multiple can be selected: `-Dname` and `Dname=definition` separated by a comma. That is, the optional argument must begin with `-D` and if the option `definition` value is not provided, it is assumed to be `1`.

The block parameter window which appears after double-clicking on the AI Engine kernel block is same irrespective of whether the kernel is Window type or Stream type. But the function configuration parameters changes for Window and Stream type. To edit the code in the MATLAB editor, click **Edit** (immediately after the browse button).

Importing Window-Based Kernels

As explained in Data Accessing Mechanisms, the size of the input and output data blocks for window-based access depends on the specified window size. Model Composer supports the following windows-based input and output data types as interfaces to the AI Engine kernel block.

- `input_window_<Type>`
- `output_window_<Type>`

<Type>	Complexity	Signedness
int8	Real	Signed
int16	Real	Signed
int32	Real	Signed
int64	Real	Signed
uint8	Real	Unsigned
uint16	Real	Unsigned
uint32	Real	Unsigned
uint64	Real	Unsigned
cint16	Complex	Signed
cint32	Complex	Signed
float	Real	N/A
cfloat	Complex	N/A

As an example, to import a simple kernel with a window-based interface, the following `simple.h` header file defines the `add_kernel` function with two input windows and one output window of type `int16`.

Simple.h

```
#ifndef __ADD_KERNEL_H__
#define __ADD_KERNEL_H__

#include <adf.h>
#define NUM_SAMPLES 4
void add_kernel(input_window_int16 * in1,input_window_int16 * in2,
output_window_int16 * outw);

#endif
```

The kernel (`simple.cc`) is defined as follows. It processes a `sum` operation on `in1` and `in2` and produces output on `outw`.

```
#include "simple.h"
void add_kernel(input_window_int16 * in1,input_window_int16 * in2,
output_window_int16 * outw)
{
    int16 temp1,temp2,temp_out;
    for (unsigned i=0; i<NUM_SAMPLES; i++) {
        window_readincr(in1,temp1);
        window_readincr(in2,temp2);
        temp_out = temp1 + temp2;
        window_writeincr(outw,temp_out);
    }
}
```

TIP: *Although not required, the following recommendations are useful for reusability and faster compilation.*

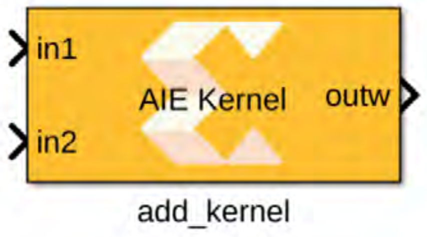
- Define each kernel in its own source file.
- Organize kernels by creating directories for header files and source files separately.
- Kernel source files should include all relevant header files to allow for independent compilation.

To import the `add_kernel` function as a block in a Model Composer design, double-click the **AIE Kernel** block and update parameters as follows:

- **Kernel header file:** `kernels/include/simple.h`
- **Kernel function:** `add_kernel`
- **Kernel Init function:** Leave empty
- **Kernel source file:** `kernels/source/simple.cc`
- **Kernel search path:** Leave empty
- **Preprocessor options:** Leave empty

When you click the **Import** button in the block parameters GUI, the tool parses the function signature in the header file and updates the AI Engine kernel block GUI interface as shown in the following figure.

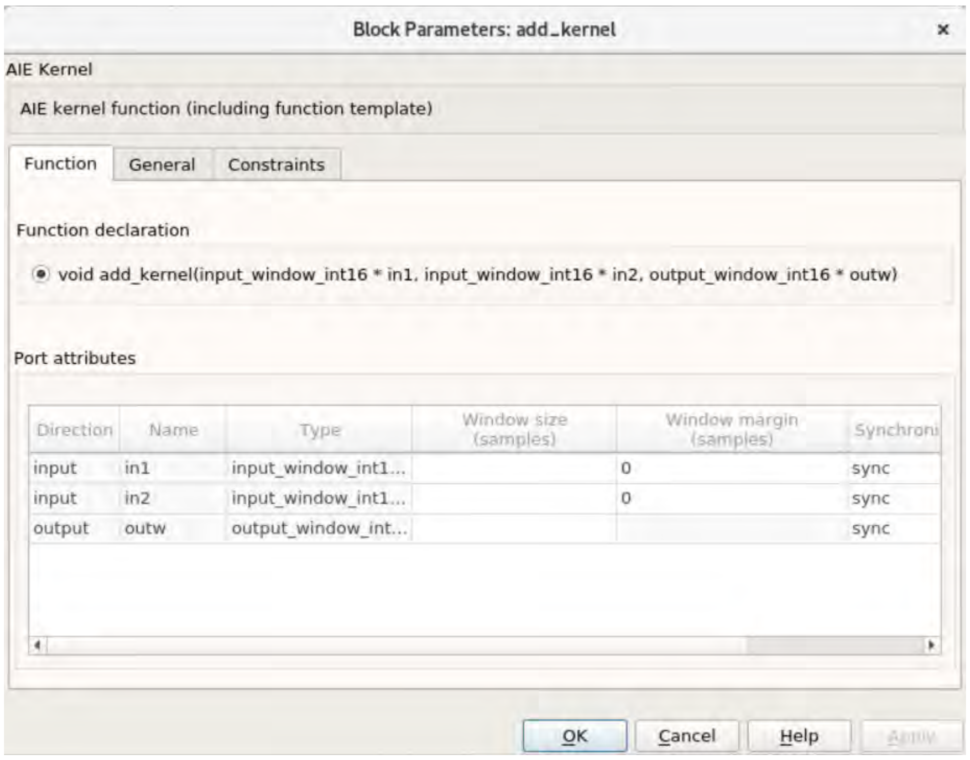
Figure 214: **AI Engine Kernel (updated)**



After the AI Engine kernel block is added to the Simulink editor, input and output ports are not present. But, after adding the kernel parameters in the GUI, the block is updated with two input ports and one output port with block name matching the imported kernel function.

After a successful import, the Function tab GUI displays automatically, providing user-editable configuration parameters. You can quickly review the function definition of the imported kernel function and the port names with directions.

Figure 215: **Function Tab**


Block Parameters: add_kernel
×
AIE Kernel
AIE kernel function (including function template)
Function
General
Constraints
OK
Cancel
Help
Apply

Function declaration

(●) void add_kernel(input_window_int16 * in1, input_window_int16 * in2, output_window_int16 * outw)

Port attributes

Direction	Name	Type	Window size (samples)	Window margin (samples)	Synchroni
input	in1	input_window_int1...		0	sync
input	in2	input_window_int1...		0	sync
output	outw	output_window_int...			sync

Appropriate values should be entered in the Function tab for Window size and Window margin (see the previous figure).

Setting the Window Margin Value

As explained in Data Accessing Mechanisms, window margin is the overlapping of input data samples. Model Composer accepts, Window margin value in terms of the number of samples. The values given in the Window margin fields should be multiple of 32 bytes.

For example, if your input data type is `int16` which is 2 bytes. The minimum Window margin value that is accepted is 16 samples (16*2). The other values that are accepted can be 32,48, 64 and so on.

Another example. If your input is of type `cint32` which is 8 bytes (real 4 bytes and 4 imaginary bytes). In this case the minimum window margin value that is accepted is `4`. Because, 4 * 8 bytes gives 32 bytes. The other values that are accepted can be 8, 12, 16, and so on.

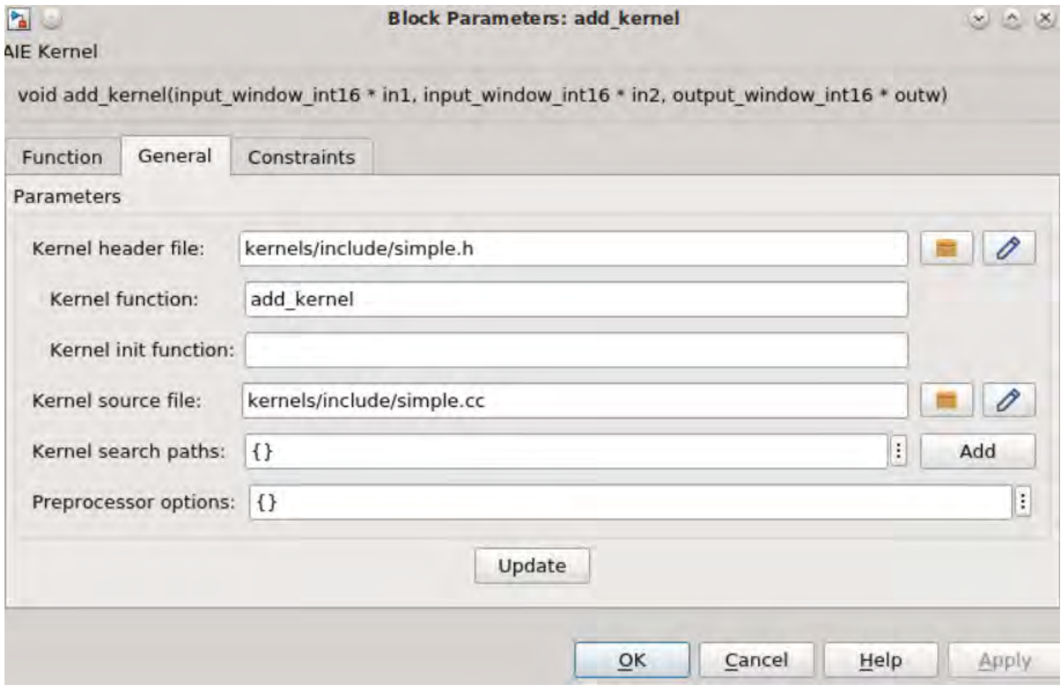
The following table provides details on the parameters and description for each parameter.

Table 17: **Window Port Parameters**

Parameter Name	Criticality	Description
Window size	Mandatory	• Window size is required for each port (argument) of the kernel function. • The value represents the number of samples (elements). • The window size must be a positive integer value. • Window size should be a multiple of 16 bytes.
Window margin	Mandatory	• Window margin is required for each input port (argument) of the kernel function. • The value represents the number of samples (elements). • The window margin must be a non-negative integer. • The window margin should be a multiple of 32 bytes.
Synchronicity	Mandatory	• The Synchronicity value options available are `sync` and `async`. • Port synchronicity is set to `sync` by default. You can optionally change it `async`.

After the successful import of kernel functions, the **Import push** button label in the General tab changes to **Update** enabling further updates of block parameters. You can change the source code of the kernel function even after importing the block without requiring a re-import. However, if you change the function signature, or the parameters to the function, then you will need to click **Update** in the **General** tab to apply changes.

Figure 216: **General Tab**


Block Parameters: add_kernel

AIE Kernel

void add_kernel(input_window_int16 * in1, input_window_int16 * in2, output_window_int16 * outw)

Kernel header file: kernels/include/simple.h

Kernel function: add_kernel

Kernel init function:

Kernel source file: kernels/include/simple.cc

Kernel search paths: {}

Preprocessor options: {}

Importing Stream-Based Kernels

As explained in Data Accessing Mechanisms, stream-based kernels access data streams in a sample-by-sample fashion. Model Composer supports the following stream-based input and output data types as interfaces to the AI Engine kernel block.

- `input_stream_<TYPE>`
- `output_window_<TYPE>`

<Type>	Complexity	Signedness
int8	Real	Signed
int16	Real	Signed
int32	Real	Signed
int64	Real	Signed
uint8	Real	Unsigned
uint16	Real	Unsigned
uint32	Real	Unsigned
uint64	Real	Unsigned
cint16	Complex	Signed
cint32	Complex	Signed
float	Real	N/A
cfloat	Complex	N/A
accfloat	Real	N/A

<Type>	Complexity	Signedness
caccfloat	Complex	N/A

As an example, to import a simple kernel with a stream-based interface, the following `simple.h` header file declares the `simple_comp` function with one input stream and one output stream.

```
#ifndef __COMPLEX_KERNEL_H__
#define __COMPLEX_KERNEL_H__
#include <adf.h>
  void simple_comp(input_stream_cint16 * in, output_stream_cint16 * out);
#endif //__COMPLEX_KERNEL_H__
```

and the function is defined in `simple.cc`.

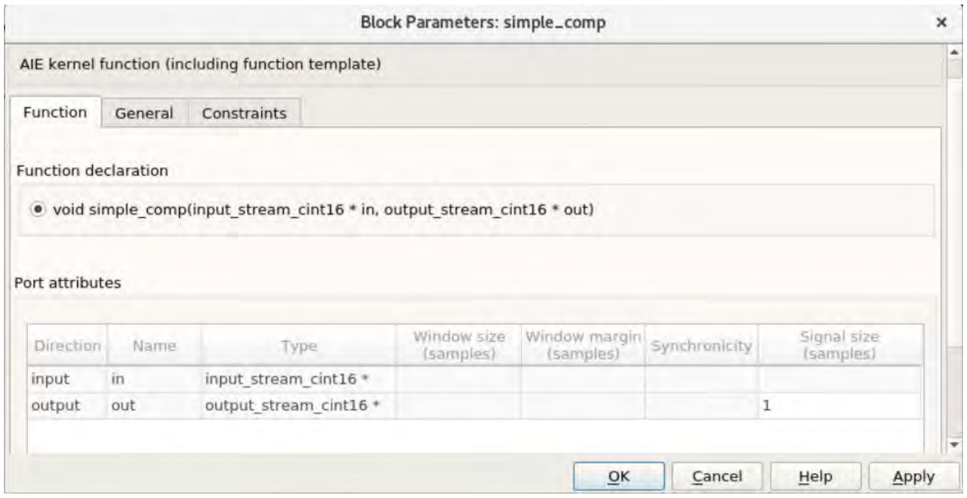
```
#include "simple.h"
void simple_comp(input_stream_cint16 * in, output_stream_cint16 * out) {
  cint16 c1, c2;

  for (unsigned i=0; i<NUM_SAMPLES; i++) {
  c1 = readincr(in);
  c2.real = c1.real+c1.imag;
  c2.imag = c1.real-c1.imag;
  writeincr(out, c2);
}
}
```

Note: See the *Versal ACAP AI Engine Programming Environment User Guide* (UG1076) for details of the `readincr()` and `writeincr()` APIs.

Although, the function arguments for window-based and stream-based kernels are different, the procedure for importing the stream-based kernel is the same.

After a successful import, the Function tab GUI displays automatically. You can quickly review the function definition and ports as shown in the following figure.

Figure 217: **Function Tab**


Block Parameters: simple_comp

AIE kernel function (including function template)

Function | General | Constraints

Function declaration

● void simple_comp(input_stream_cint16 * in, output_stream_cint16 * out)

Port attributes

Direction	Name	Type	Window size (samples)	Window margin (samples)	Synchronicity	Signal size (samples)
input	in	input_stream_cint16 *				
output	out	output_stream_cint16 *				1

The following table provides details on the parameters and a description for each parameter.

Table 18: **Stream Port Parameters**

Parameter Name	Description
Signal size	• This parameter represents the size of the output signal and should be set to a value greater than or equal to the maximum nnumber of samples that are produced during any invocation of the kernel.

In the General tab, the **Import** button changes to **Update**, enabling further updates of block parameters.

Model Composer also supports cascade stream connections between two AI Engine processors.

An AI Engine kernel can use incoming stream connections as follows:

- `Input_stream_acc48`
- `Input_stream_cacc48`

Similarly, a kernel can use the outgoing cascade stream connections as follows:

- `output_stream_acc48`
- `output_stream_cacc48`

In Model Composer, a cascade stream port is represented as a 48-bit fixed point signal (`x_sfix48`) that can be either complex or real.

Consider the following example, where a cascade output stream of one kernel is connected to the cascade input stream of another kernel.

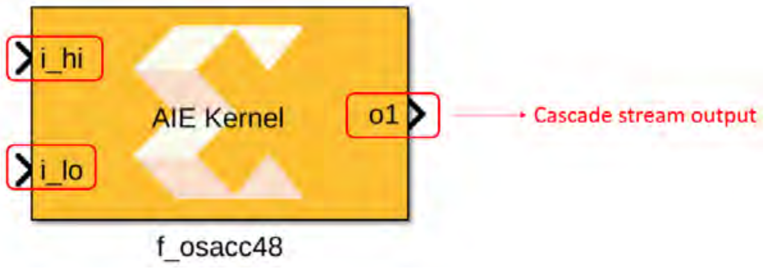
```
#ifndef __CASCADE_KERNELS_H__
#define __CASCADE_KERNELS_H__
void f_osacc48(input_window_int32 *i_hi,
               input_window_int16 *i_lo,
               output_stream_acc48 *o1);
#endif
```

The kernel function `f_osacc48` has two input windows: `i_hi` and `i_lo`, and one cascade stream output: `o1`.

***Note*:** This kernel function includes both window-based ports and stream-based ports.

After importing this kernel function, the AI Engine kernel block is as shown in the following figure.

Figure 218: **AI Engine Kernel after Import**

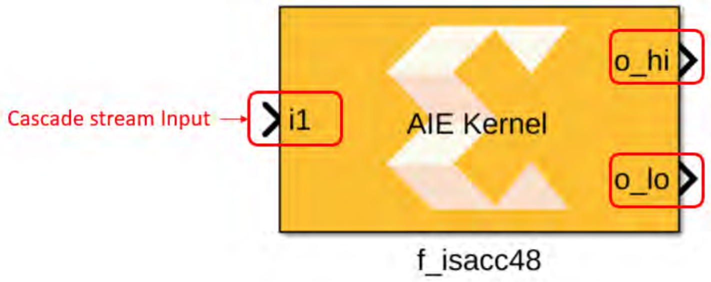


Consider another kernel function `f_isacc48`, which has one cascade stream input: `i1`, and two output windows: `o_hi` and `o_lo`.

```
#ifndef __CASCADE_KERNELS_H__
#define __CASCADE_KERNELS_H__
void f_isacc48(input_stream_acc48 *i1,
               output_window_int32 *o_hi,
               output_window_int16 *o_lo);
#endif
```

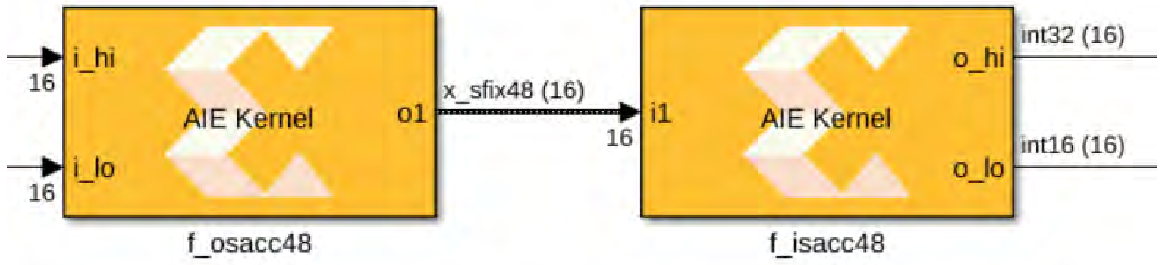
After importing the second kernel function, the AI Engine kernel block is as shown in the following figure.

Figure 219: **AI Engine Kernel (Second Kernel Function)**



Now the two kernels can be connected to form a cascade connection using the cascade stream output of block `f_osacc48` and the cascade stream input of block `f_isacc48`. This is shown in the following figure.

Figure 220: **Connected Kernels (Cascade Connection)**



Importing an AI Engine Kernel with Run-Time Parameters

Model Composer supports importing AI Engine kernels with run-time parameters in kernel functions alongside window and stream types. The following table lists the scalar data types that can be passed as run-time parameters.

<Type>	Complexity	Signedness
int8	Real	Signed
int16	Real	Signed
int32	Real	Signed
int64	Real	Signed
uint8	Real	Unsigned
uint16	Real	Unsigned
uint32	Real	Unsigned
uint64	Real	Unsigned
cint16	Complex	Signed
cint32	Complex	Signed
float	Real	N/A

<Type>	Complexity	Signedness
cfloat	Complex	N/A

Implicit ports are inferred for each parameter (scalar and vector data types) in the function argument. The following table describes the type of port inferred for each function argument.

Formal Parameter	Port Class
T	Input
Const T	Input
T &	Inout
Const T &	Input
Const T (&) [..]	Input
T (&)[..]	Inout

In the following example, the `simple_rtp` function has two real-time parameters. Notice the function argument `select` which is passed by value, and argument `weight` which is passed by reference.

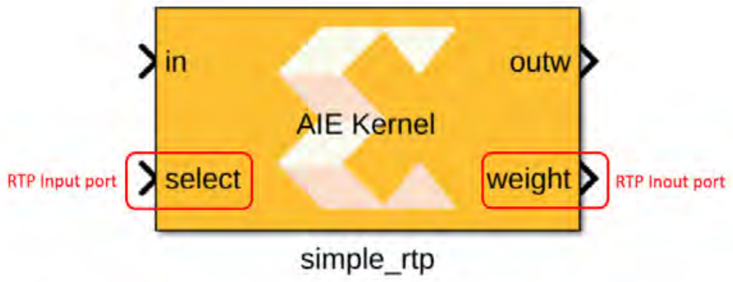
```
#ifndef __RTP_KERNEL_H__
#define __RTP_KERNEL_H__

void simple_rtp(input_window_cint16 * in,output_window_cint16 * outw, int32
&weight, int32 select);

#endif //__RTP_KERNEL_H__
```

When imported for the above function, the AI Engine kernel block looks as shown in the following figure. In Model Composer, the `inout` port appears as the `output` port on the AI Engine kernel block.

Note**:** Model Composer ignores the `inout` RTP ports during code generation and only considers them for Simulink simulation. (i.e., they will not be read from the PS).

Because RTPs are used alongside the window and stream ports, the procedure for importing the kernel function remains the same. When the above kernel function with RTPs are imported, the AI Engine kernel block looks as shown in the following figure.

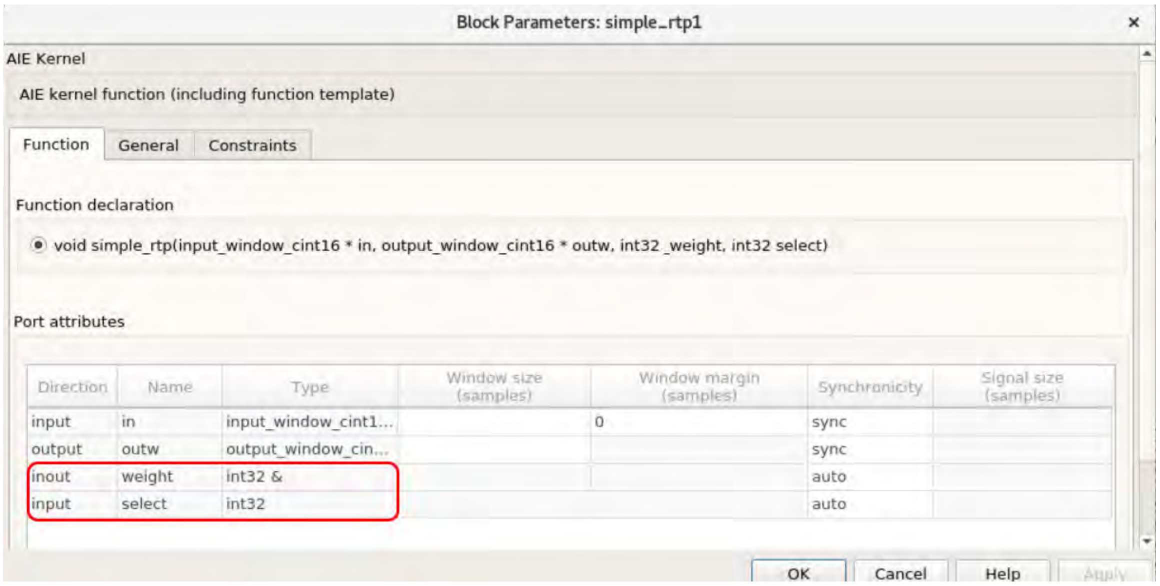
Figure 221: **AI Engine Kernel (RTPs)**



Notice that the AI Engine kernel block name (`simple_rtp`) is same as AI Engine kernel function name.

After a successful import, the Function tab GUI displays automatically. You can quickly review the function definition and run-time parameter ports as shown.

Figure 222: **Function Tab**



Block Parameters: simple_rtp1

AIE Kernel

AIE kernel function (including function template)

Function | General | Constraints

Function declaration

● void simple_rtp(input_window_cint16 * in, output_window_cint16 * outw, int32 _weight, int32 select)

Port attributes

Direction	Name	Type	Window size (samples)	Window margin (samples)	Synchronicity	Signal size (samples)
input	in	input_window_cint1...		0	sync	
output	outw	output_window_cin...			sync	
inout	weight	int32 &			auto	
input	select	int32			auto	

Port synchronicity is the only parameter that is specific to RTPs. The following table provides details about the valid synchronicity of the Destination RTP input port with respect to the Source RTP inout port. The default port synchronicity is set to 'auto'.

Table 19: **Valid Synchronicity**

Source RTP Inout Port	Destination RTP input Port
auto	async
sync	auto
sync	sync

Table 19: **Valid Synchronicity** *(cont'd)*

Source RTP Inout Port	Destination RTP input Port
async	async

If the source RTP inout port is set to 'auto' then the destination RTP input port should be 'async'. Similarly for other combinations. Model Composer throws an appropriate error when you try to use any combination which is not specified in the previous table.

Importing an AI Engine Kernel with Function Template

You may require a generic function that can be used for different datatypes. Using templates, you can pass a datatype as a parameter and Model Composer supports importing an AI Engine Kernel with a function template. To do this, use the same AIE Kernel block used earlier to import the ordinary C++ functions.

As an example to import the kernel function with templates, consider below header file `kernel.h`, containing the declaration of a function template. Here, the template has typename template parameter T, and a non-type (integral) template parameter N.

`kernel.h`

```
#ifndef _AIE_TEMP_KERNELS_H_
#define _AIE_TEMP_KERNELS_H_

#include <adf.h>
template<typename T, int N>
void myFunc(input_window<T>  *i1,
            output_window<T> *o1
            );

#endif // ifndef _AIE_TEMP_KERNELS_H
```

The definition of the function template is in the source file `kernel.cpp` as shown below.

`kernel.cpp`

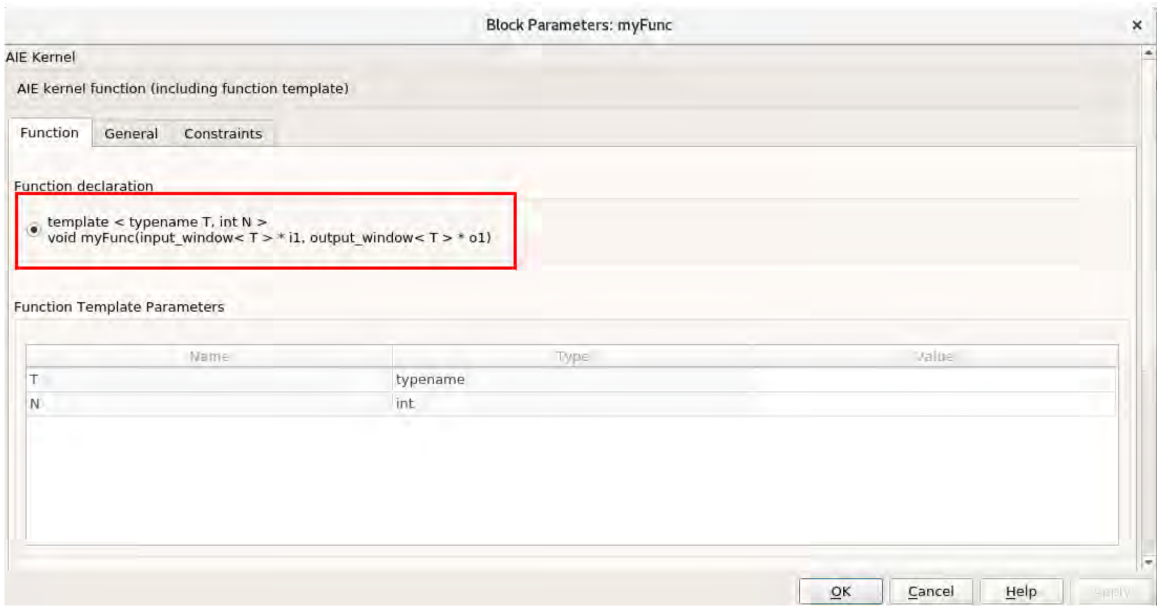
```
#include "kernel.h"

template<typename T, int N>
void
myFunc(input_window<T>  *i1,
       output_window<T> *o1
       )
{
   for(int i = 0;i<8;i++)
   {
       T val = window_readincr(i1);
       val *= N;
       window_writeincr(o1, val);
    }
}
```

Notice the usage of the non-type template parameter 'N' in the kernel output computation. To import the template function as a block into Model Composer, double-click the AIE Kernel block and update the parameters as follows.

- **Kernel header file:** `kernels/include/kernel.h`
- **Kernel function:** `myFunc`
- **Kernel Init function:** Leave empty
- **Kernel source file:** `kernels/source/kernel.cpp`
- **Kernel search path:** Leave empty
- **Preprocessor options:** Leave empty

When you click the **Import** button in the block parameters GUI, the **Function** tab displays automatically. Enter the values of a template type parameter 'T' and a template non-type parameter of integral type within the Function Template Parameter section as shown in the following figure. Double-click the appropriate editable field and enter the values. You can also review the declaration of the template function in the Function declaration section.

Figure 223: **AIE Kernel: Function declaration**



template < typename T, int N >
void myFunc(input_window< T > * i1, output_window< T > * o1)

Name	Type	Value
T	typename	
N	int	

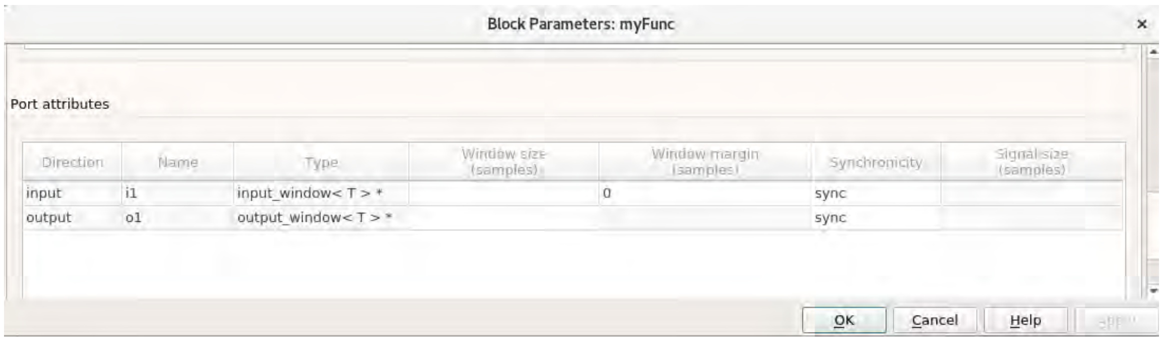
The following typenames are supported as type template parameters:

- int8,int16,int32,int64
- uint8,uint16,uint32,uint64
- float, double

- cint16, cint32, cfloat

Scroll down to the Port attributes section in the Function tab and enter appropriate user-editable configuration parameters as shown.

Figure 224: **AIE Kernel: Port attributes**

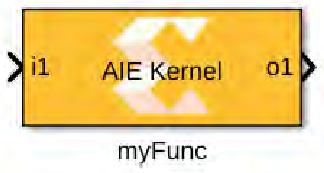


Port attributes

Direction	Name	Type	Window size (samples)	Window margin (samples)	Synchronicity	Signal size (samples)
input	i1	input_window< T > *		0	sync	
output	o1	output_window< T > *			sync	

After entering the appropriate values in the Function tab, click **Apply**. Notice the updated interface of the AIE kernel block GUI as shown in the following figure.

Figure 225: **AIE Kernel: Updated**



Template Specialization

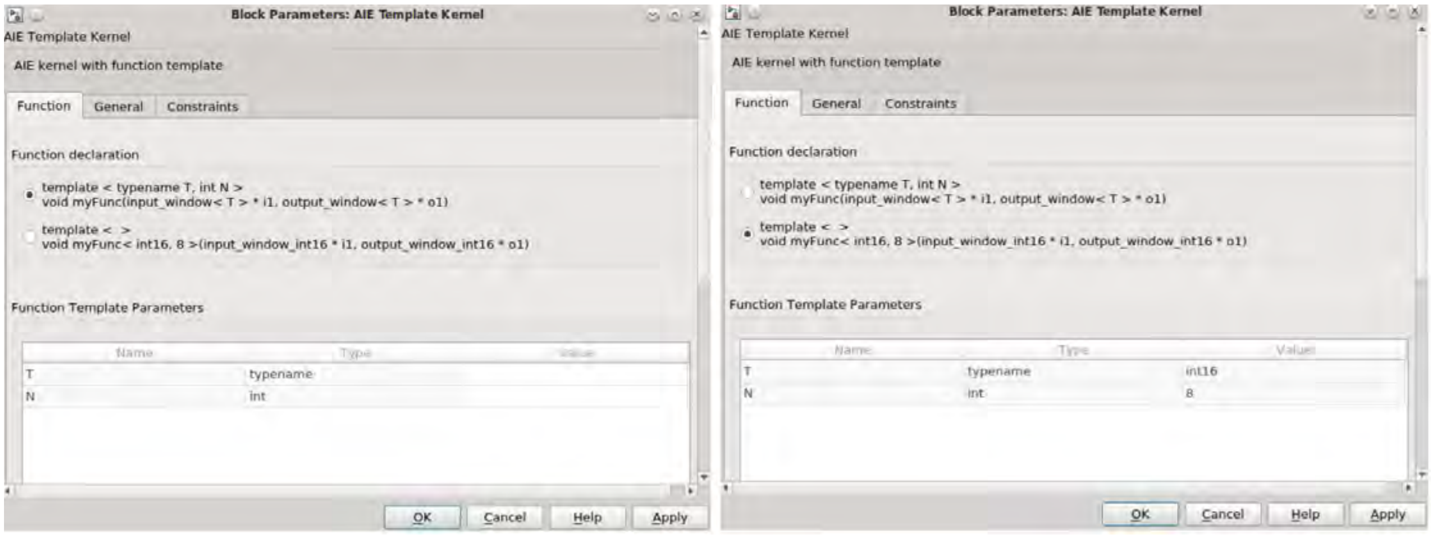
For cases where you want to override the default template implementation to handle a particular type in a different way, Model Composer supports template specialization. Consider the following example where a function `myFunc` has one specialized version declared to implement an `int16` datatype along with a generic template function.

`kernel.cpp`

```
#ifndef _AIE_CLASS_KERNELS_H_
#define _AIE_CLASS_KERNELS_H_

#include <adf.h>
template<typename T, int N>
void myFunc(input_window<T>  *i1,
            output_window<T> *o1
            );
template<>
void myFunc<int16,8>(input_window<int16>  *i1,
       output_window<int16> *o1);

#endif
```

When you try to import the kernel `myFunc` as a block into Model Composer using the AIE Template Kernel block, the Function tab in the block GUI parameter looks as shown. If you select the function variant corresponding to the base template, the Function Template Parameters table shows the values corresponding to that. If you select the specialization variant instead, the table shows the values of the template parameters of that specialization. You cannot change these values.

Figure 226: **AIE Template Kernel: Functional declaration Options**


Block Parameters: AIE Template Kernel

AIE Template Kernel

AIE kernel with function template

Function | General | Constraints

Function declaration

[✓ template < typename T, int N >
void myFunc(input_window< T > * i1, output_window< T > * o1)]

template < >
void myFunc< int16, 8 >(input_window_int16 * i1, output_window_int16 * o1)

Function Template Parameters

Name	Type	Value
T	typename	
N	int	

OK | Cancel | Help | Apply

Block Parameters: AIE Template Kernel

AIE Template Kernel

AIE kernel with function template

Function | General | Constraints

Function declaration

template < typename T, int N >
void myFunc(input_window< T > * i1, output_window< T > * o1)

[✓ template < >
void myFunc< int16, 8 >(input_window_int16 * i1, output_window_int16 * o1)]

Function Template Parameters

Name	Type	Value
T	typename	int16
N	int	8

OK | Cancel | Help | Apply

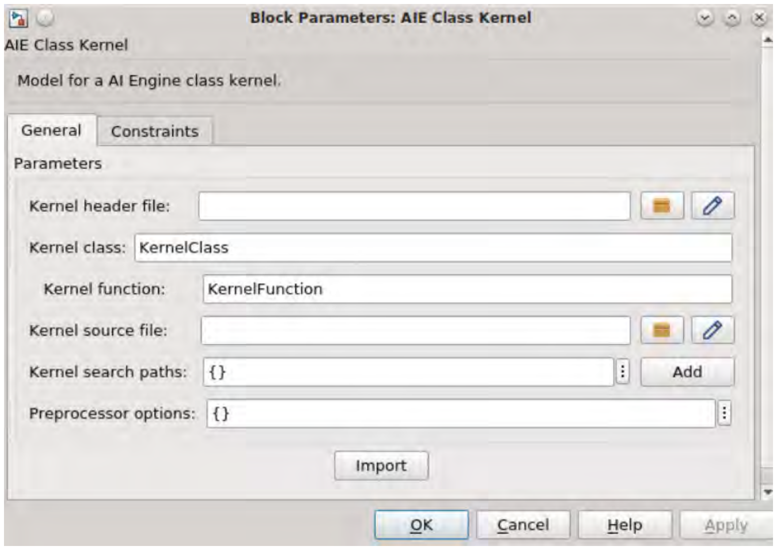
Class-Based Kernels

Model Composer supports importing the C++ kernel class to have constructor parameters for specifying parameter values. You need to use anAI Engine class kernel block from the AI Engine Library as shown.

Figure 227: **AIE Class Kernel**

Double-clicking the block symbol displays the parameters of the AI Engine class kernel block as shown in the following figure.

Figure 228: **AIE Class Kernel: Block Parameters**


Block Parameters: AIE Class Kernel

AIE Class Kernel

Model for a AI Engine class kernel.

General | Constraints

Parameters

Kernel header file:

Kernel class: KernelClass

Kernel function: KernelFunction

Kernel source file:

Kernel search paths: {}

Preprocessor options: {}

The block mask parameters need to be updated in order to import the kernel function as a block. The following table provides details on the parameters and description for each parameter.

Parameter Name	Parameter Type	Criticality	Description
Kernel header file	String	Mandatory	Name of the header file that contains the kernel class and registerKernelClass method declarations The string could be just the file name, a relative path to the file or an absolute path of the file. Use the browse button to select the file. This field does not accept environmental variables.
Kernel class	String	Mandatory	Name of the kernel class which contains member variables and kernel member function.
Kernel function	String	Mandatory	Name of the kernel member function for which the block is to be created. This function should be registered using the registerKernelClass method in kernel header file.
Kernel init function	String	Optional	Name of the initialization function used by the kernel function.
Kernel source file	String	String	Name of the source file that contains the kernel member function definition and non-default constructor parameter values are specified. The string could be the file name, a relative path to the file or an absolute path of the file. This field does not accepts environmental variables.
Kernel search paths	Vector of Strings	Optional	If the kernel header file or the kernel source file are not found using the value provided through 'Kernel header file' or 'Kernel source file' fields, respectively, then the paths provided through 'Kernel search paths' are used to find the files. This parameter allows use of environment variables while specifying paths for the kernel header file and the kernel source file. The environment variable can be used in either `${ENV}` or `$ENV` format.

Parameter Name	Parameter Type	Criticality	Description
Preprocessor options		Optional	Optional preprocessor arguments for downstream compilation with specific preprocessor options. The following two preprocessor option formats are accepted and multiple can be selected: `-Dname` and `-Dname=definition` separated by a comma. That is, the optional argument must begin with `-D` and if the the option *definition* value is not provided, it is assumed to be `1`.

The AI Engine class kernel block supports all the kernel functions that a normal AI Engine kernel block can support and the block parameter window which appears after double-clicking on the AI Engine class kernel block is the same irrespective of whether the kernel member function is Window-based or Stream-based. To edit the header file or source file, you can click the **Edit** button (immediately after the browse button).

Kernel Class with Default Constructor

As an example, to import the C++ class kernel with the default constructor, consider the following `simple.h` header file that defines the kernel class `simple_class` with the default constructor.

`simple.h`

```
#include "adf.h"

class simple_class
{
private:
    int16 val;
    int16 numSamples;

public:
    simple_class();

    void mulBytwo(input_window_int16* in, output_window_int16* out);

    static void registerKernelClass()
    {
        REGISTER_FUNCTION(simple_class::mulBytwo);
    }
};
```

RECOMMENDED: *It is highly recommended to define the body of the kernel and class constructors in `.cpp` file.*

It is necessary to register the kernel function using the `registerKernelClass()` method. More than one class kernel can be declared in a header file and each class should contain separate `registerKernelClass()` methods. Only one function can be registered per class and Model Composer can import only functions that are registered using REGISTER_FUNCTION().

The Kernel function is defined in `simple.cpp` as shown below.

`simple.cpp`

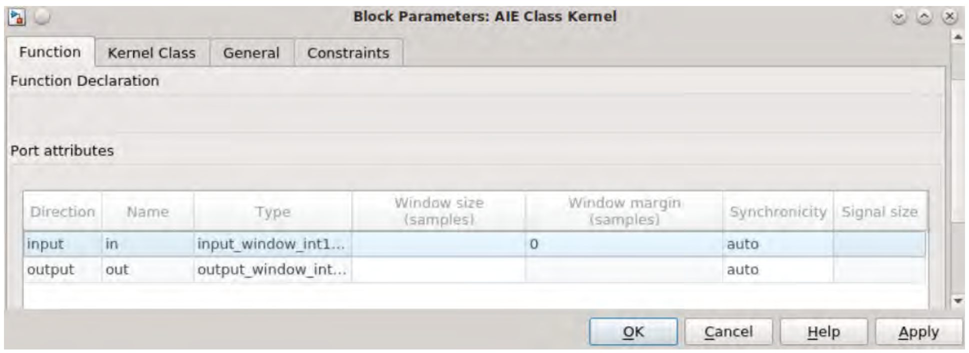
```
#include "simple.h"

simple_class::Simple_class()
{
    val = 24;
    numSamples = 8;
}

void simple_class::mulBytwo(input_window_int16* in, output_window_int16*
out)
{
    for (int i=0; i<numSamples; i++)
    {
        int16 value = window_readincr(in);
        window_writeincr(out, (in*2)+val);
    }
}
```

To import the `mulBytwo` function as a block in a Model Composer design, double click the **AIE class kernel** block and update the parameters as follows.

- **Kernel header file:** `kernels/include/simple.h`
- **Kernel class:** `simple_class`
- **Kernel function:** `mulBytwo`
- **Kernel Init function:** Leave empty
- **Kernel source file:** `kernels/source/simple_kernel.cpp`
- **Kernel search path:** Leave empty
- **Preprocessor options:** Leave empty

Click the **Import** button in the block parameters GUI. After successful import, the Function tab displays. This provides user-editable configuration parameters as shown in the following figure.

Figure 229: **AIE Class Kernel: Block Parameters**



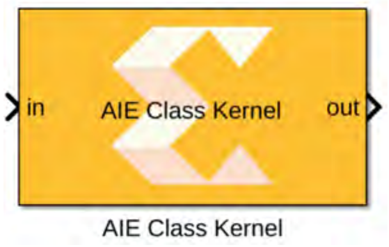
Function Declaration

Port attributes

Direction	Name	Type	Window size (samples)	Window margin (samples)	Synchronicity	Signal size
input	in	input_window_int1...		0	auto	
output	out	output_window_int...			auto	

After entering the appropriate values in the Function tab, click **Apply** to see the updated interface of the AI Engine class kernel block GUI as shown.

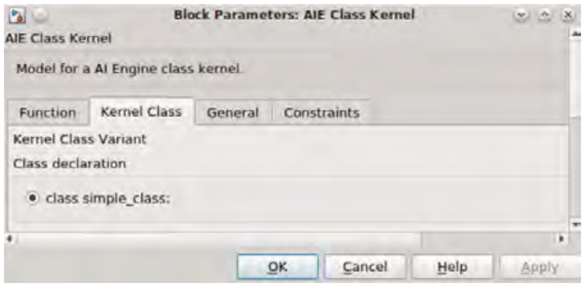
Figure 230: **AIE Class Kernel after Update**



You can quickly review the function declaration of the imported kernel function and the port names with directions, from the Function tab.

Click the **Kernel Class** tab to observe the class declaration as shown in the following figure.

Figure 231: **Kernel Class Tab: Class Declaration**


Block Parameters: AIE Class Kernel

AIE Class Kernel

Model for a AI Engine class kernel

Kernel Class Variant

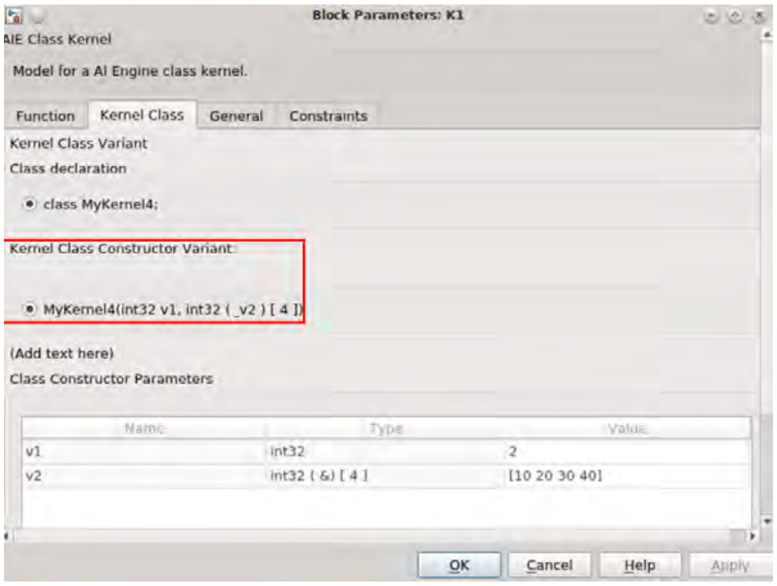
Class declaration

● class simple_class;

Class Kernels with Parameterized Constructors

Default constructors do not take any arguments and have no parameters. However, it is possible to pass arguments to the constructors and Model Composer supports importing the class kernels with parameterized constructors. Both scalar and vector arguments can be passed to constructors. You can assign values to these parameters from the **Kernel Class** tab in the AI Engine Class Kernel block as shown in the following figure. You can also observe the parameterized constructor declaration in Kernel Class Constructor Variant section.

Figure 232: **Kernel Class: Parameter Values**



AIE Class Kernel

Model for a AI Engine class kernel.

Kernel Class Variant

Class declaration

(●) class MyKernel4;

Kernel Class Constructor Variant:

(●) MyKernel4(int32 v1, int32 (_v2) [4])

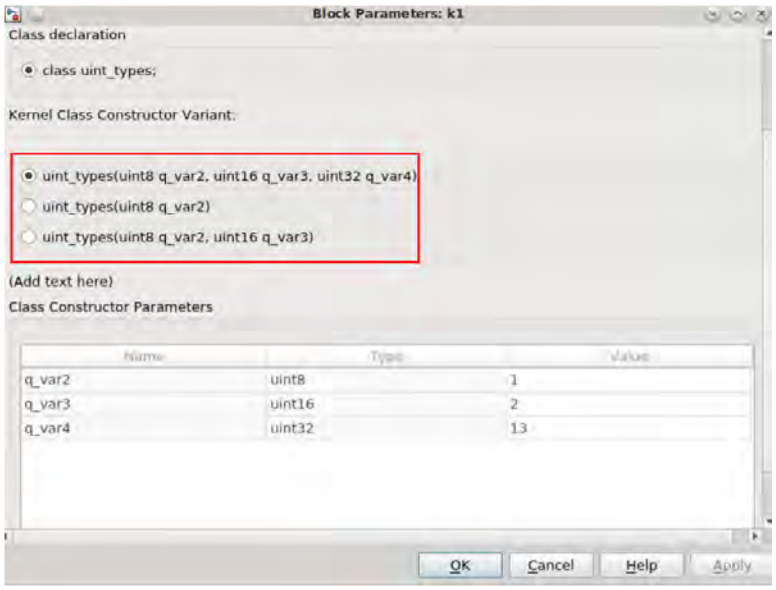
(Add text here)

Class Constructor Parameters

Name	Type	Value
v1	int32	2
v2	int32 (&) [4]	[10 20 30 40]

If you have multiple variants of Kernel Class Constructors, you can choose one of them and pass values to the constructor arguments accordingly.

Figure 233: **Kernel Class: Multiple Values**



Class declaration

(●) class uint_types;

Kernel Class Constructor Variant:

(●) uint_types(uint8 q_var2, uint16 q_var3, uint32 q_var4)
( ) uint_types(uint8 q_var2)
( ) uint_types(uint8 q_var2, uint16 q_var3)

(Add text here)

Class Constructor Parameters

Name	Type	Value
q_var2	uint8	1
q_var3	uint16	2
q_var4	uint32	13

Constructor with Reference to an Array

Consider the following example that declares the constructor with reference to an array as argument.

`fir.h`

```
class FIR
{
    private:
        int32 (&coeffs)[NUM_COEFFS];
        int32 tapDelayLine[NUM_COEFFS];
        uint32 numSamples;
    public:
        FIR(int32(&coefficients)[NUM_COEFFS], uint32 samples);
        void filter(input_window_int32* in, output_window_int32* out);
        static void registerKernelClass()
        {
            REGISTER_FUNCTION(FIR::filter);
            REGISTER_PARAMETER(coeffs);
        }
};
```

`fir.cpp`

```
#include "fir.h"
FIR::FIR(int32(&coefficients)[NUM_COEFFS], uint32 samples)
: coeffs(coefficients)
{
    for (int i = 0; i < NUM_COEFFS; i++)
    tapDelayLine[i] = 0;
    numSamples = samples;
}
void FIR::filter(input_window_int32* in, output_window_int32* out)
{
    ...
}
```

Here, member variable `coeffs` is an `int32 (&)[NUM_COEFFS]` data type. The constructor initializer `coeffs(coefficients)` initializes `coeffs` to the reference to an array allocated externally to the class object. To let the aiecompiler know that the `coeffs` member variable is intended to be allocated by the compiler, you must use REGISTER_PARAMETER to register an array reference member variable inside the `registerKernelClass()` method. The `aiecompiler` throws an appropriate error if the constructors with reference to an array are not registered.

Kernel with Class Templates

You may require a class implementation that remains the same for all classes but the data types vary. Model Composer supports importing the kernels with class templates using the AIE Class Kernel block. Consider the following example which declares the class template in `kernel.h`.

`kernel.h`

```
#ifndef _AIE_CLASS_KERNELS_H_
#define _AIE_CLASS_KERNELS_H_
#include <adf.h>

template<typename T, int N>
class MyKernel {
   int m_count;
public:
   MyKernel();
   void myFunc(input_stream<T>  *i1,
               output_stream<T> *o1,
               output_stream<int> *o2);

    static void registerKernelClass()
    {
        REGISTER_FUNCTION(MyKernel::myFunc);
    }

};
#endif
```

In this case, the default constructor initializer `m_count(N)` initializes `m_count` with template parameter `N` as shown in te following `kernel.cpp` code.

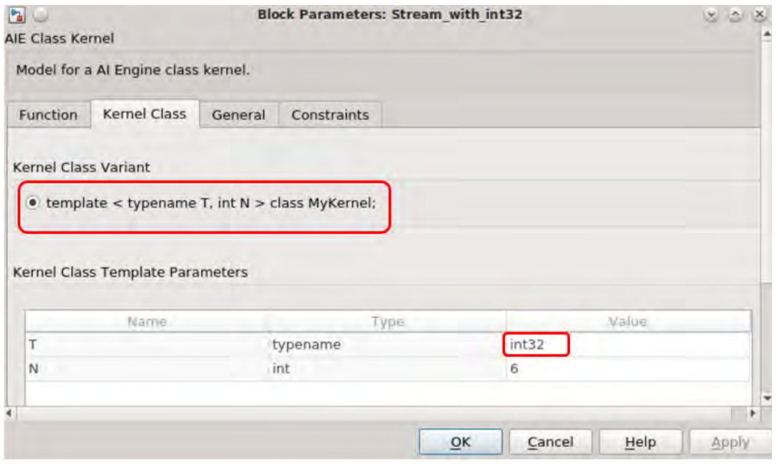
`kernel.cpp`

```
#include "kernel.h"

template<typename T, int N>
MyKernel<T,N>::MyKernel()
   : m_count(N)
{
}

template<typename T, int N>
void
MyKernel<T,N>::myFunc(input_stream<T>  *i1,
                      output_stream<T> *o1,
                      output_stream<int> *o2)
{
   put_ms(0, get_ss(0) * N);
   ++m_count;
   writeincr(o2, m_count);
}
```

After successfully importing the kernel with class template using the AIE Class Kernel block, the Function tab displays. Here you can enter appropriate values in the user-editable configuration parameters. Click **Apply** to see the updated interface of the AI Engine class block GUI.

Redirect to the **Kernel Class** tab in the block parameters GUI to review the template class declaration from the kernel class variant. In the **Kernel Class** tab, you can enter the value of a template type parameter 'T' and a template non-type parameter of integral type as shown.

Figure 234: **Kernel Class Template**


Block Parameters: Stream_with_int32

AIE Class Kernel

Model for a AI Engine class kernel.

Kernel Class Variant

● template < typename T, int N > class MyKernel;

Kernel Class Template Parameters

Name	Type	Value
T	typename	int32
N	int	6

IMPORTANT!

1. Template type parameters can be any valid window, stream, or RTP datatypes.
2. Only a value that has an integral type is supported for template non-type parameters.
3. MATLAB variables can be used to specify non-type template parameters.

Template Specialization

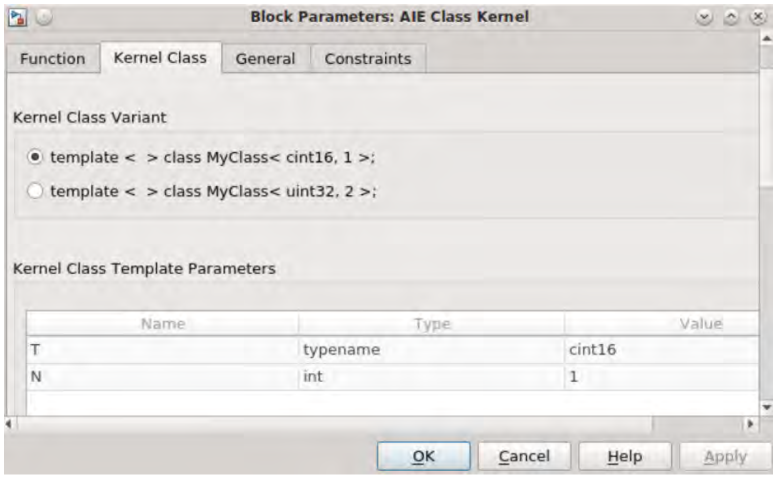
For cases when you need to override the default template implementation to handle a particular type in a different way, Model Composer supports template specialization. Consider the following example where a class `MyClass` has two different interfaces than the generic `MyClass`. One specialized version is declared to implement the `cint16` datatype and other version to implement the `uint32` datatype.

`template_specialization.h`

```
#include <adf.h>
template<typename T,int N>
class MyClass
{
};

template<>
class MyClass<cint16,1> {
   int m_count;
   int16 var_1;
   int16 var_2;
   int16 var_3;
   uint16 var_4;
public:
   MyClass();
   MyClass(int16 q_var1,int16 q_var2,int16 q_var3,uint16 q_var4);
   MyClass(int16 q_var1,int16 q_var2);
   MyClass(int16 q_var1,int16 q_var2,int16 q_var3);

   void func_mem(input_stream<cint16>  *i1,
                  output_stream<cint16> *o1,
                  output_stream<int> *o2);

     static void registerKernelClass()
     {
         REGISTER_FUNCTION(MyClass::func_mem);
     }

};

template<>
class MyClass<uint32,2> {
   int m_count;
   int16 var;
public:
   MyClass(uint32 q_var1);

   void func_mem(input_stream<uint32>  *i1,
                 output_stream<uint32> *o1,
                 output_stream<int> *o2);

     static void registerKernelClass()
     {
         REGISTER_FUNCTION(MyClass::func_mem);
     }

};
```

You can see that two functions are registered separately in two specialized classes. When you try to import the kernel `func_mem` as a block into Model Composer using the AIE Class kernel block, the **Kernel Class** tab in block GUI parameters looks as shown.

Figure 235: **Class Variant**



Block Parameters: AIE Class Kernel

Function | Kernel Class | General | Constraints

Kernel Class Variant

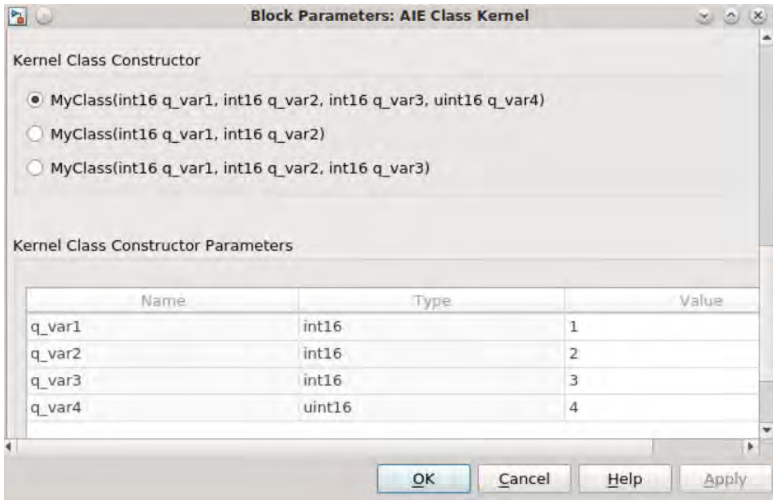
(●) template < > class MyClass< cint16, 1 >;
( ) template < > class MyClass< uint32, 2 >;

Kernel Class Template Parameters

Name	Type	Value
T	typename	cint16
N	int	1

After selecting one of the Kernel Class Variants from the list, the Class Template Parameters update accordingly. The list of Kernel Class Constructors for the corresponding class variant, is updated and you can select from the list (see the following figure).

Figure 236: **Kernel Class Constructors**



Kernel Class Constructor

(●) MyClass(int16 q_var1, int16 q_var2, int16 q_var3, uint16 q_var4)
( ) MyClass(int16 q_var1, int16 q_var2)
( ) MyClass(int16 q_var1, int16 q_var2, int16 q_var3)

Kernel Class Constructor Parameters

Name	Type	Value
q_var1	int16	1
q_var2	int16	2
q_var3	int16	3
q_var4	uint16	4

***Note*:** MATLAB variables can be used to specify the values of Kernel Class Constructor Parameters.

Template Partial Specialization

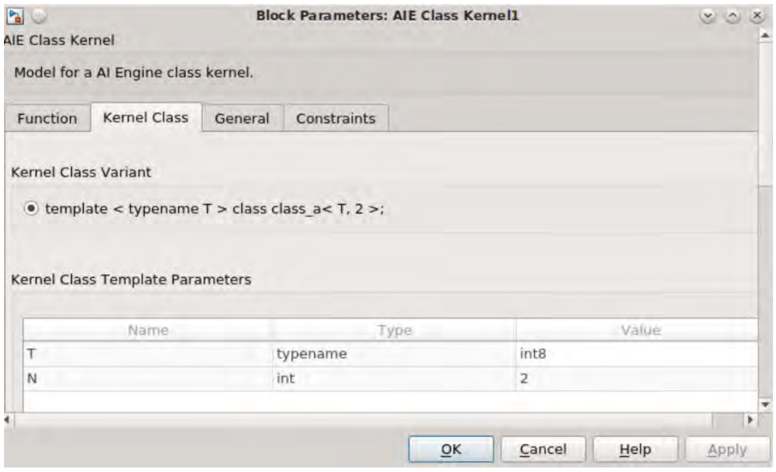
For cases where you write a template that specializes one template parameter and still allows some parameterization, you can use the template partial specialization. Model Composer allows you to import the class kernels with partial specialization using the AIE Class Kernel block. Consider the following example where a class `class_a` is partially specialized with a non-type template parameter.

`partial_specialization.h`

```
#include <adf.h>
template<typename T,int N>
class class_a
{
};

template<typename T>
class class_a<T,2> {
   int m_count;
   T var;
public:
   class_a(T q_var1);

   void func_mem(input_stream<T>  *i1,
                       output_stream<T> *o1,
                       output_stream<int> *o2);

    static void registerKernelClass()
    {
        REGISTER_FUNCTION(class_a::func_mem);
    }

};
```

Notice that the function `func_mem` is registered in `registerKernelClass()` method. When you try to import the kernel function as a block into Model Composer using the AIE Class Kernel block, the Kernel Class tab in block GUI parameters looks as shown.

Figure 237: **Block Parameters: AIE Class Kernel**


Block Parameters: AIE Class Kernel1

AIE Class Kernel

Model for a AI Engine class kernel.

Kernel Class Variant

(●) template < typename T > class class_a< T, 2 >;

Kernel Class Template Parameters

Name	Type	Value
T	typename	int8
N	int	2

Because the class is partially specialized with a non-type template parameter, you cannot edit the parameter 'N' from the Kernel Class Template Parameters. However, the value of the template type parameter can be a modified to any valid datatype.

Kernels with Namespaces

Model Composer supports importing kernels declared in a namespace. For both templatized and non-templatized kernels you need to qualify the kernel function with the namespace.

Consider the following examples where the non-templatized kernel function is qualified with the namespace `ns1` and templatized kernel function is qualified with the namespace `ns2`.

`Kernel.h`

```
namespace ns1 {
void myFunc_1(input_stream<int32> * restrict i1,
            output_stream<int32> * restrict o1);
} // namespace ns1
```

`templateKernel.h`

```
namespace ns2 {
template<typename T,int size>
void myFunc_2(input_stream<int32> * restrict i1,
            output_stream<int32> * restrict o1);
} // namespace ns2
```

To import the above functions using the AIE Kernel and AIE Template Kernel blocks, the Kernel functio' parameter in the GUI block parameters should be updated as follows:

- **Kernel Function:** `ns1::myFunc_1` (Non -templatized function)
- **Kernel Function:** `ns2::myFunc _2` (Templatized function)

If you have a class kernel declared in a namespace, then only the kernel class field should be qualified, and not the kernel function.

For example, consider the following kernel class which is qualified with the namespace `ns3`.

`class_kernel.h`

```
namespace ns3 {
#include "adf.h"

class simple_class
{
private:
    int16 val;
    int16 numSamples;

public:
    simple_class();

    void func_q(input_window_int16* in, output_window_int16* out);

    static void registerKernelClass()
    {
        REGISTER_FUNCTION(simple_class::func_q);
    }
};
} // namespace ns3
```

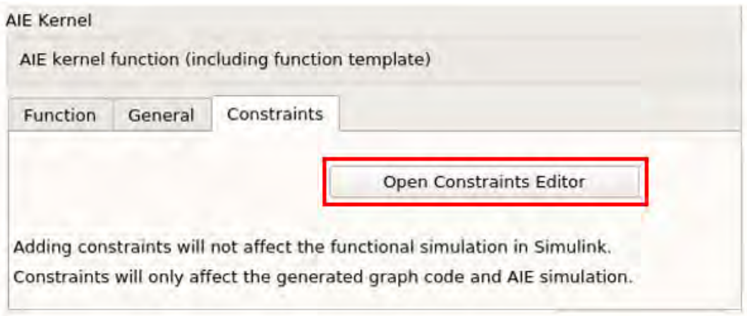
To Import the kernel function `func_q` as a block, the Kernel Class and Kernel function parameters in the AIE Class Kernel block should be updated as follows:

- **Kernel class:** `ns3::simple_class`
- **Kernel function:** `sfunc_q`

Specifying Constraints

Constraints are user-defined properties for graph nodes which provide additional information to the compiler. Model Composer provides a mechanism to specify these constraints from within the kernel import or graph import blocks in the AI Engine library. During graph code generation, all these constraints automatically appear inside the graph code. The following figure shows the Constraints tab of the AIE Kernel block highlighting the Open Constriants Editor button in it.

Figure 238: **Constraints**

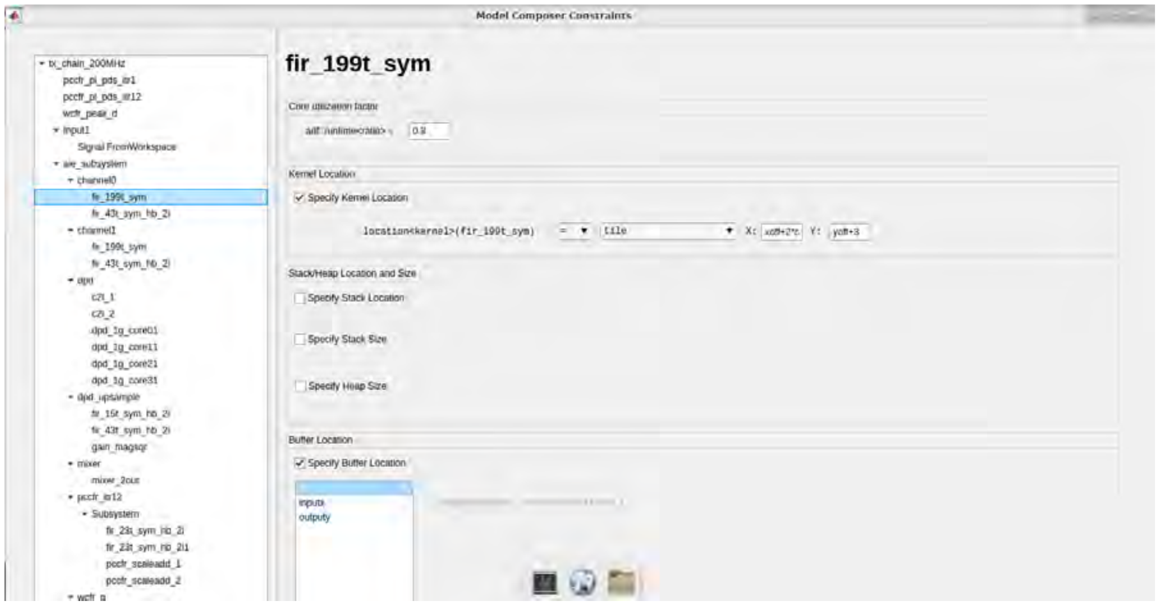


AIE kernel function (including function template)

Adding constraints will not affect the functional simulation in Simulink.
Constraints will only affect the generated graph code and AIE simulation.

Clicking **Open Constraints Editor** opens the Model Composer Constraints window. Here you can specify various constraints such as core utilization factor, kernel location, buffer location, stack/ heap location, and size as shown in the following figure.

Figure 239: **Model Composer Constraints**



fir_199t_sym

Core utilization factor

aie::runtime<ratio> = 0.9

Kernel Location

[✓] Specify Kernel Location

location<kernel>(fir_199t_sym) = tile X: x0+2*c Y: y0+3

Stack/Heap Location and Size

[ ] Specify Stack Location

[ ] Specify Stack Size

[ ] Specify Heap Size

Buffer Location

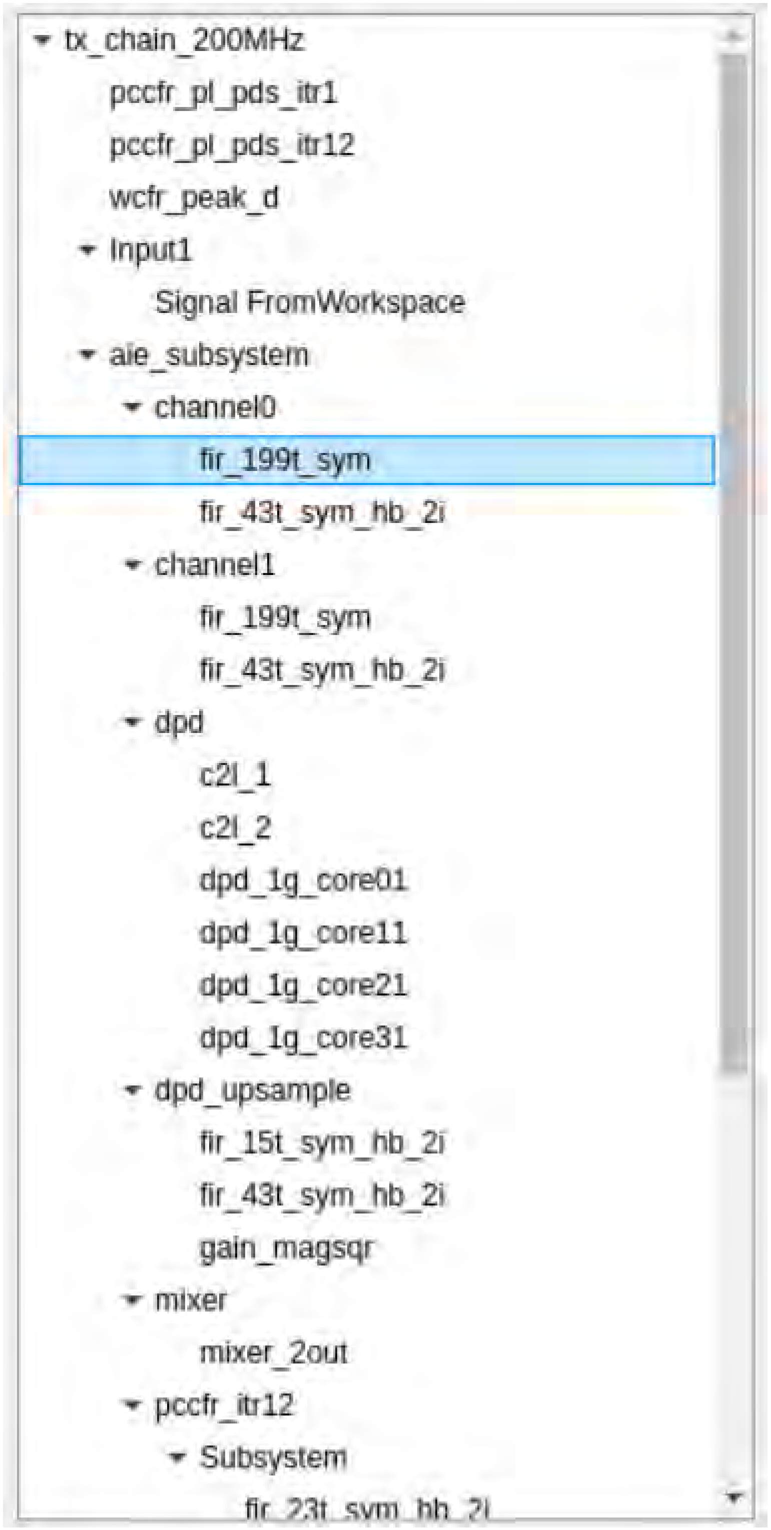
[✓] Specify Buffer Location

inputx
outputy

A similar constraints tab is available for all blocks in the Xilinx Toolbox/AI Engine/User-Defined Functions library and the constraint editor reflects the constraints available for that particular block. Adding these constraints will not affect Simulink simulation as they are only used for generating the graph code and AIE simulation.

When you open the constraints editor from any particular kernel/graph import block, the Model Composer Constraints window allows you to specify the constraints for that particular kernel. However, you can switch between all the kernels/graphs available in your design from the navigation panel on the left side and specify the constraints accordingly as shown in the following figure. In this way, you can avoid opening each kernel/graph block separately to specify constraints.

Figure 240: **Navigation Panel in Constraints Editor**

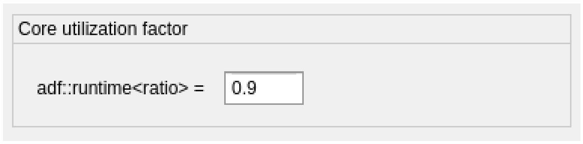


The remainder of this section discusses the types of constraints that Model Composer supports.

Core utilization factor (runtime<ratio>)

The core utilization factor ratio is specified as the ratio of the function run time compared to the cycle budget and must be between 0 and 1. The cycle budget is the number of instruction cycles a function can take to either consume data from its input or to produce a block of data on its output.

You can specify the core utilization factor in the Model Composer Constraints editor window as shown.


Core utilization factor

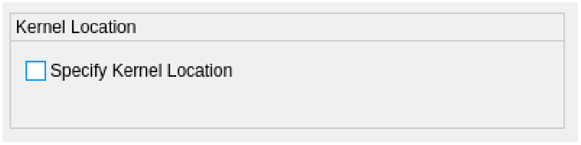
adf::runtime<ratio> = 0.9

Kernel Location

When building large graphs with multiple subgraphs, it is sometimes useful to control the exact mapping of kernels to AI Engines, either relative to other kernels or in an absolute sense. Specifying location constraints provides a powerful mechanism to create a robust, scalable, and predictable mapping of your graph onto the AI Engine array. It also reduces the choices for the mapper to try, which can considerably speed up the mapper.

The kernel location constraint can be specified from the Model Composer Constraints editor window as shown.

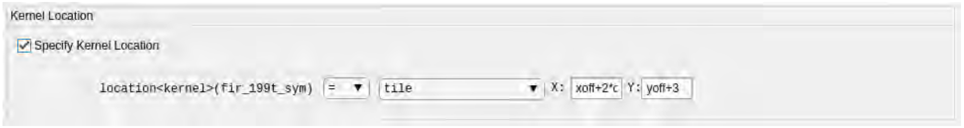
- By default, the option **Specify Kernel Location** is deselected. Enable this to specify the constraint.


Kernel Location

☐ Specify Kernel Location

You can choose to either:

- Constrain a kernel to be placed on a specified AI Engine tile.


Kernel Location

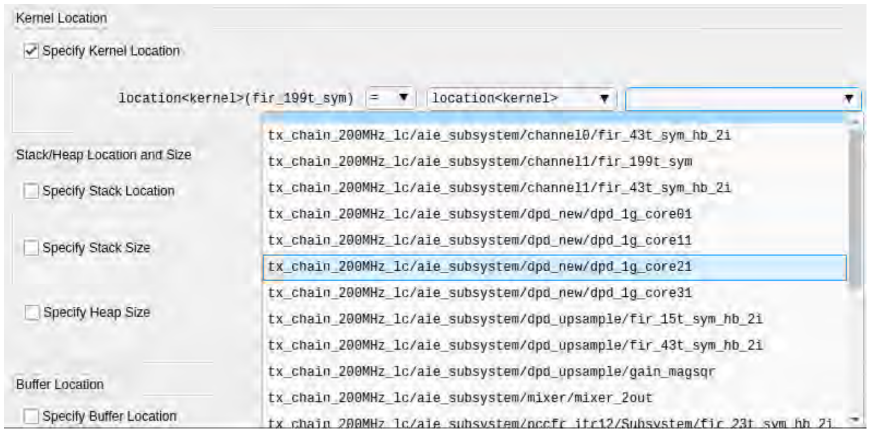
☑ Specify Kernel Location

location<kernel>(fir_199t_sym) = tile X: xoff+2*c Y: yoff+3

IMPORTANT! *MATLAB variables can be used to specify the kernel locations.*

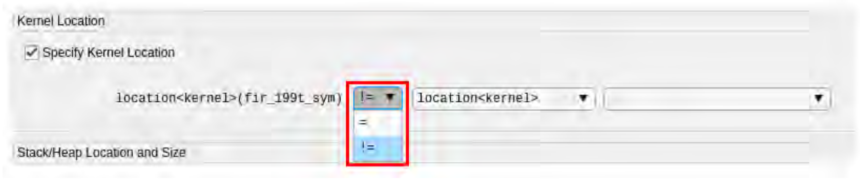
Or

- Constrain two kernels to be placed relatively on the same AI Engine. This forces them to be sequenced in topological order and be able to share memory buffers without synchronization. As shown, you can select the kernel you want to place relatively on the AI Engine from the drop down.



```
Kernel Location
[✓] Specify Kernel Location
    location<kernel>(fir_199t_sym)  =  location<kernel>
        tx_chain_200MHz_lc/aie_subsystem/channel0/fir_43t_sym_hb_2i
        tx_chain_200MHz_lc/aie_subsystem/channel1/fir_199t_sym
        tx_chain_200MHz_lc/aie_subsystem/channel1/fir_43t_sym_hb_2i
        tx_chain_200MHz_lc/aie_subsystem/dpd_new/dpd_1g_core01
        tx_chain_200MHz_lc/aie_subsystem/dpd_new/dpd_1g_core11
        tx_chain_200MHz_lc/aie_subsystem/dpd_new/dpd_1g_core21
        tx_chain_200MHz_lc/aie_subsystem/dpd_new/dpd_1g_core31
        tx_chain_200MHz_lc/aie_subsystem/dpd_upsample/fir_15t_sym_hb_2i
        tx_chain_200MHz_lc/aie_subsystem/dpd_upsample/fir_43t_sym_hb_2i
        tx_chain_200MHz_lc/aie_subsystem/dpd_upsample/gain_magsqr
        tx_chain_200MHz_lc/aie_subsystem/mixer/mixer_2out
        tx_chain_200MHz_lc/aie_subsystem/pccfr_itr12/Subsystem/fir_23t_sym_hb_2i
Stack/Heap Location and Size
[ ] Specify Stack Location
[ ] Specify Stack Size
[ ] Specify Heap Size
Buffer Location
[ ] Specify Buffer Location
```

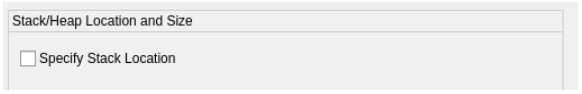
- Model Composer also supports specifying two kernels, say k1 and k2. These should not be mapped to the same AI Engine.



```
Kernel Location
[✓] Specify Kernel Location
    location<kernel>(fir_199t_sym)  !=  location<kernel>
                                    =
                                    !=
Stack/Heap Location and Size
```

Stack Location

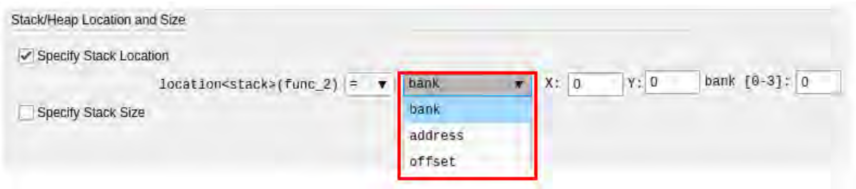
The stack location constraint is used to specify the location of the system memory (stack and heap) of the AI Engine where the specified kernel is mapped. This provides the mechanism to constrain the location of the system memory with respect to other buffers used by that kernel. By default, the option **Specify Stack Location** is deselected. Enable this to specify the constraint.



```
Stack/Heap Location and Size
[ ] Specify Stack Location
```

You can specify the stack location by pointing to a:

- Specific data memory bank on an AI Engine tile. The bank ID is relative to the tile and can take values 0,1,2,3.
- Specific data memory address on an AI Engine tile. The offset address is relative to the tile starting at zero with a maximum value of 32768 (32K).
- Specific data memory address offset. The offset address is between 0 and 32768 (32K) and is relative to a tile allocated by the compiler.

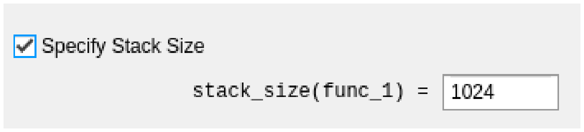


```
Stack/Heap Location and Size
[✓] Specify Stack Location
    location<stack>(func_2)  =  bank   X: 0   Y: 0   bank [0-3]: 0
                                bank
                                address
                                offset
[ ] Specify Stack Size
```

Model Composer also supports specifying the stack location of the kernel where it should not be mapped to a particular bank, address, or offset.

Stack Size

This constraint allows you to set the stack size for an individual kernel. By default the option **Specify Stack Size** is deselected. Enable this to specify the constraint. The default value is 1024.



☑ Specify Stack Size

stack_size(func_1) = 1024

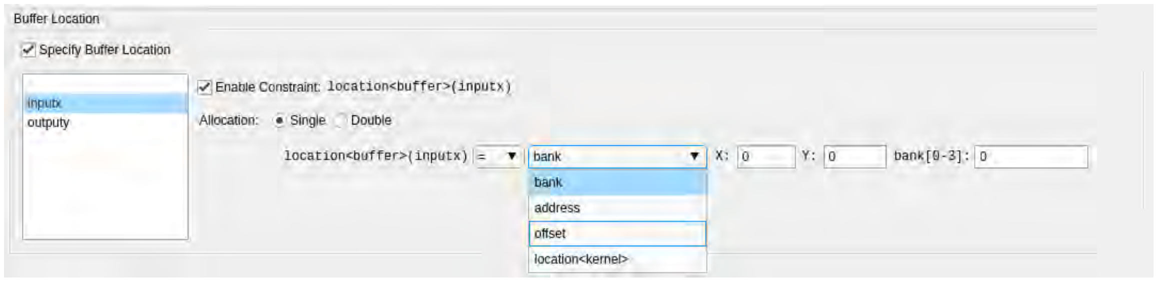
Buffer Location

The AI Engine compiler attempts to automatically allocate buffers for windows, lookup tables, and run-time parameters in the most efficient manner possible. However, you might want to explicitly control their placement in memory. Similar to the kernels shown previously in this section, buffers inferred on a kernel port can also be constrained to be mapped to specific tiles, banks, or even address offsets using location constraints.

Buffer Location

☐ Specify Buffer Location

- By default, the option **Specify Buffer Location** is deselected. When you enable this option, the kernel ports that can be constrained are displayed. Buffer locations are only allowed on window kernel ports.
- You can click on each individual port and enable constraint as shown in the following figure.
- You can use the **Allocation** option to specify a single or double buffer constraint on a window port. By default, a window port is double buffered.
- For a single buffer allocation, you can choose to constrain the buffer location by pointing to a:
  - Specific data memory bank on an AI Engine tile. The bank ID is relative to the tile and can take values 0,1,2,3.
  - Specific data memory address on an AI Engine tile. The offset address is relative to the tile starting at zero with a maximum value of 32768 (32K).
  - Specific data memory address offset. The offset address is between 0 and 32768 (32K) and is relative to a tile allocated by the compiler.
  - Specific buffer location to be on the same bank as that of one or more other port buffers. This ensures that the buffer can be accessed by other kernel without requiring a DMA.


Buffer Location

☑ Specify Buffer Location

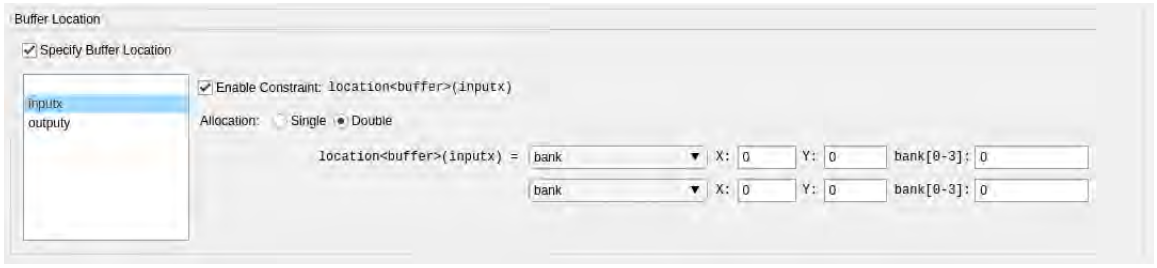
inputx
outputy

☑ Enable Constraint: `location<buffer>(inputx)`

Allocation: ● Single ○ Double

`location<buffer>(inputx)` = bank X: 0 Y: 0 bank[0-3]: 0

- You can constrain the location of double buffers attached to a port that are to be placed on a specific address or a bankid.


Buffer Location

☑ Specify Buffer Location

inputx
outputy

☑ Enable Constraint: `location<buffer>(inputx)`

Allocation: ○ Single ● Double

`location<buffer>(inputx)` = bank X: 0 Y: 0 bank[0-3]: 0

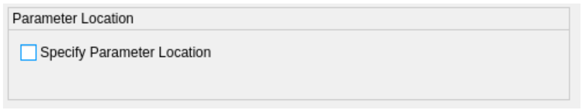
bank X: 0 Y: 0 bank[0-3]: 0

IMPORTANT!

The non-collocation constraint (i.e., specifying where the buffer should not be mapped to) is allowed only for single buffer.

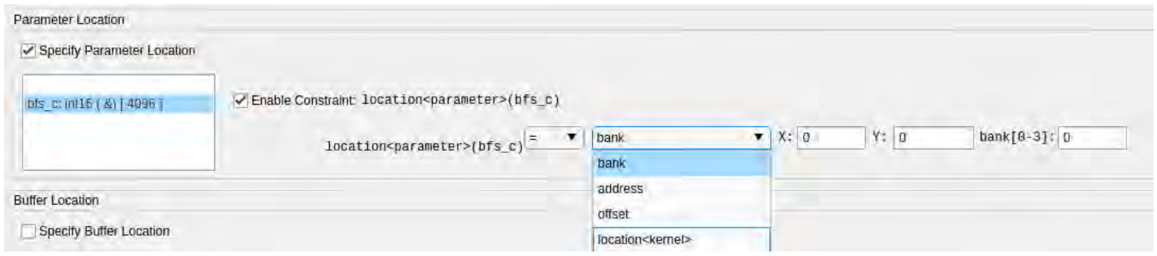
Parameter Location

This constraint allows you to set the location of the parameter array declared within the graph.


Parameter Location

☐ Specify Parameter Location

By default, the option **Specify Parameter Location** is deselected. When you enable this option, the parameters that can be constrained are displayed. You can click on each individual parameter and constrain the location of a parameter lookup table to be placed on specific address or a bankid. You can also constrain the parameter location to be on the same tile as that of some other kernel. This ensures that the buffer, or parameter array can be accessed by the other kernel without requiring a DMA.


Parameter Location

☑ Specify Parameter Location

bfs_c: int16 (&) [4096]

☑ Enable Constraint: `location<parameter>(bfs_c)`

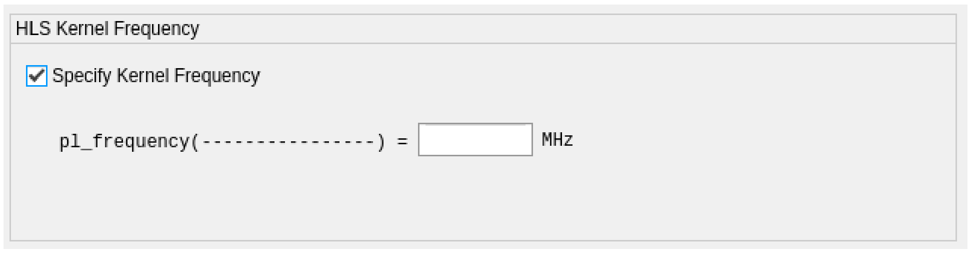
`location<parameter>(bfs_c)` = bank X: 0 Y: 0 bank[0-3]: 0

Buffer Location

☐ Specify Buffer Location

HLS Kernel Frequency

This constraint allows you to specify the clock frequency (in MHz) of the PL Kernel.

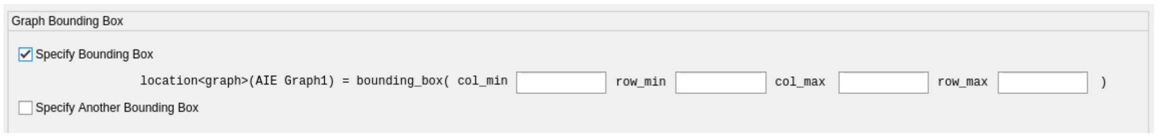

HLS Kernel Frequency

☑ Specify Kernel Frequency

`pl_frequency(----------------) =` [ ] `MHz`

Graph Bounding Box

This bounding box constraint specifies a rectangular bounding box for a graph to be placed in AI Engine tiles, between columns from `column_min` to `column_max` and rows from `row_min` to `row_max`. Multiple bounding box location constraints can be set to specify an irregular shape bounding region.


Graph Bounding Box

☑ Specify Bounding Box

`location<graph>(AIE Graph1) = bounding_box( col_min` [ ] `row_min` [ ] `col_max` [ ] `row_max` [ ] `)`

☐ Specify Another Bounding Box

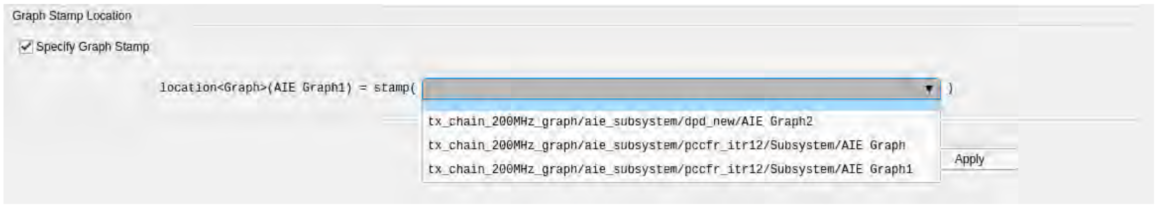
By default, the **Specify Bounding Box** option is deselected. Enable this to specify the boundary location of the graph. Further, you can also enable the **Specify Another Bounding Box** option to specify multiple bounding regions.

Graph Stamp Location

The graph stamp location constraint can be used when the same graph has multiple instances that can be constrained to the same geometry in AI Engine. There are two main advantages of using this constraint:

- When the same graph is instantiated multiple times, the throughput should be same. Because of the differences in routing, throughput might not be the exactly identical. However, it will be much closer when stamping is used.
- The run time required will be significantly less because the AI Engine compiler only solves a reference graph instead of the entire design.

By default, the **Specify Graph Stamp** option is deselected. When you enable this, Model Composer allows you to select the graph from the drop down menu for which the graph (in lhs) can be stamped.



```
location<Graph>(AIE Graph1) = stamp(                    )
tx_chain_200MHz_graph/aie_subsystem/dpd_new/AIE Graph2
tx_chain_200MHz_graph/aie_subsystem/pccfr_itr12/Subsystem/AIE Graph
tx_chain_200MHz_graph/aie_subsystem/pccfr_itr12/Subsystem/AIE Graph1
```

***Note*:** Applying constraints on Xilinx Toolbox/AI Engine/DSP library blocks is not supported and the constraint editor does not show DSPlibrary blocks in the drop-down menu.

***Note*:** When you copy an AIE block, the constraints applied on the original block may not be properly copied.

Importing AI Engine Graphs

As discussed in AI Engine Graphs, a graph is a connection of different compute kernel functions. Unlike when importing kernels, where a kernel function is imported as a block into Model Composer, in this case, graph code is imported as a block. To import the graph as block into Model Composer, you need to select the AI Engine graph block from the AI Engine library (shown in the following figure).

Figure 241: **AI Engine Graph**

Model Composer allows to connect the AI Engine graph block with the AI Engine kernel block so that the whole design can be simulated in the Simulink environment.

IMPORTANT! *It is assumed that your kernel code is already developed and that the associated kernels are properly organized.*

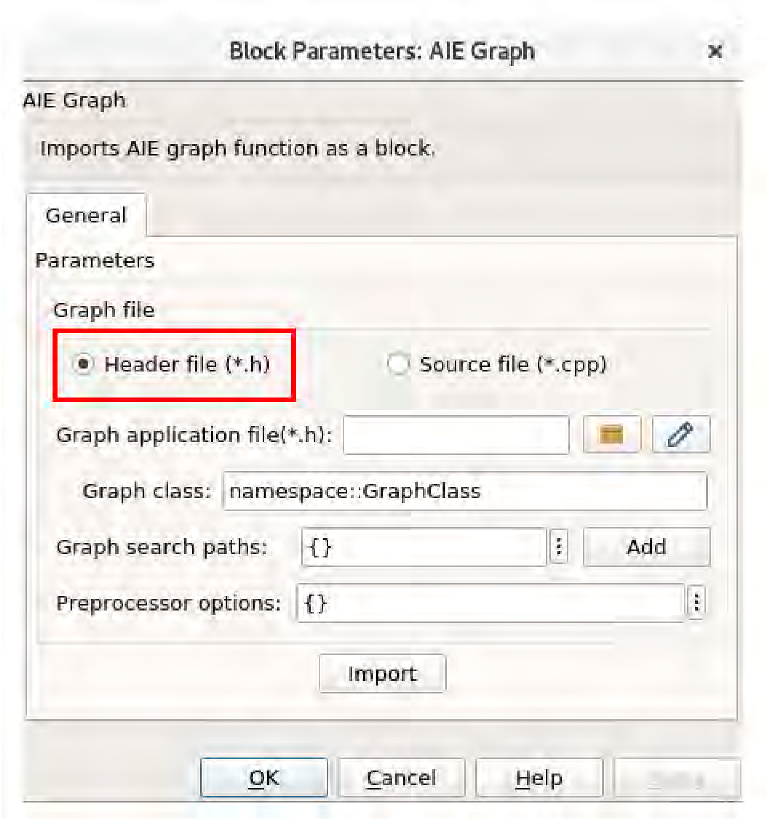
The AIE Graph block supports importing the AI Engine graph into Model Composer in two ways:

- Using the header file(`*.h`)
- Using the source file(`*.cpp`)

Using the Header File

To import the graph using the header file (`.h`), double-click the **AIE Graph** block and select the **Header file (*.h)** option in the General tab. Specify the graph header file, class, search path, and preprocessor options as shown in the following figure.

Figure 242: **AI Engine Graph Block Parameters**



Block Parameters: AIE Graph

AIE Graph

Imports AIE graph function as a block.

General

Parameters

Graph file

● Header file (*.h) ○ Source file (*.cpp)

Graph application file(*.h):

Graph class: namespace::GraphClass

Graph search paths: {}

Preprocessor options: {}

The following table provides further details including names and descriptions of each parameter.

Table 20: **AI Engine Graph Block Parameters**

Parameter Name	Parameter Type	Criticality	Description
Graph Application file (`*.h`)	String	Mandatory	Specify the file (`.h`), where the application graph class is defined and the Adaptive Data Flow (ADF) header (`adf.h`), kernel function prototypes are included.
Graph Class	String	Mandatory	Specify the name of the graph class.
Graph Search paths	Vector of strings	Mandatory	Specify search paths where header files, kernels, and other include files can be found and included for simulation. The search path `$XILINX_VITIS/adf/include` (where `adf.h` isdefined) is be included by default and does not need to be specified.

Table 20: **AI Engine Graph Block Parameters** *(cont'd)*

Parameter Name	Parameter Type	Criticality	Description
Preprocessor options		Optional	Optional preprocessor arguments for downstream compilation with specific preprocessor options. The following preprocessor option formats are accepted and multiple can be selected: '`-Dname`' and '`-Dname=definition`'. That is, the optional argument must begin with the '`-D`' string and if the option definition value is not provided, it is assumed to be 1.

`graph.h`

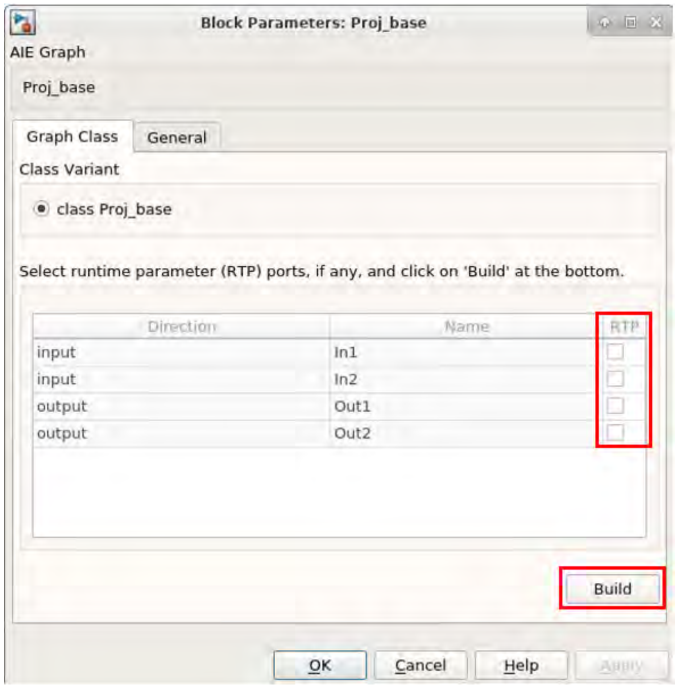
```
#ifndef __XMC_PROJ_H__
#define __XMC_PROJ_H__

#include <adf.h>
#include "simple.h"

class Proj_base : public adf::graph {
public:
   adf::kernel AIE_Kernel;

public:
   adf::input_port In1, In2;
   adf::output_port Out1, Out2;

   Proj_base() {
      // create kernel AIE_Kernel
      AIE_Kernel = adf::kernel::create(simple_comp_1);
      adf::source(AIE_Kernel) = "simple.cc";

      // create kernel constraints AIE_Kernel
      adf::runtime<ratio>( AIE_Kernel ) = 0.9;

      // create nets to specify connections
      adf::connect< adf::stream > net0 (In1, AIE_Kernel.in[0]);
      adf::connect< adf::stream > net1 (In2, AIE_Kernel.in[1]);
      adf::connect< adf::stream > net2 (AIE_Kernel.out[0], Out1);
      adf::connect< adf::stream > net3 (AIE_Kernel.out[1], Out2);
   }
};

class Proj : public adf::graph {
public:
   Proj_base mygraph;

public:
   adf::input_plio In1, In2;
   adf::output_plio Out1, Out2;

   Proj() {
      In1 = adf::input_plio::create("In1",
            adf::plio_32_bits,
            "./data/input/In1.txt");

      In2 = adf::input_plio::create("In2",
            adf::plio_32_bits,
            "./data/input/In2.txt");
```

```
        Out1 = adf::output_plio::create("Out1",
               adf::plio_32_bits,
               "Out1.txt");

        Out2 = adf::output_plio::create("Out2",
               adf::plio_32_bits,
               "Out2.txt");

        adf::connect< > (In1.out[0], mygraph.In1);
        adf::connect< > (In2.out[0], mygraph.In2);
        adf::connect< > (mygraph.Out1, Out1.in[0]);
        adf::connect< > (mygraph.Out2, Out2.in[0]);
    }
};

#endif // __XMC_PROJ_H__
```

Note: It is not allowed to import the graph class that has PLIO attributes specified. Model Composer will throw an appropriate error if you try to do so. In this case, use the `Proj_base` class to import the graph.

When the GUI parameters are updated, click the **Import** button. The tool updates the Block parameters GUI as shown. In the Graph Class tab you can see the RTP column with check boxes. You can check this ON, only if the port is a Run Time Parameter. If not, you can directly click **Build** at the bottom.

Figure 243: **Block Parameters GUI**

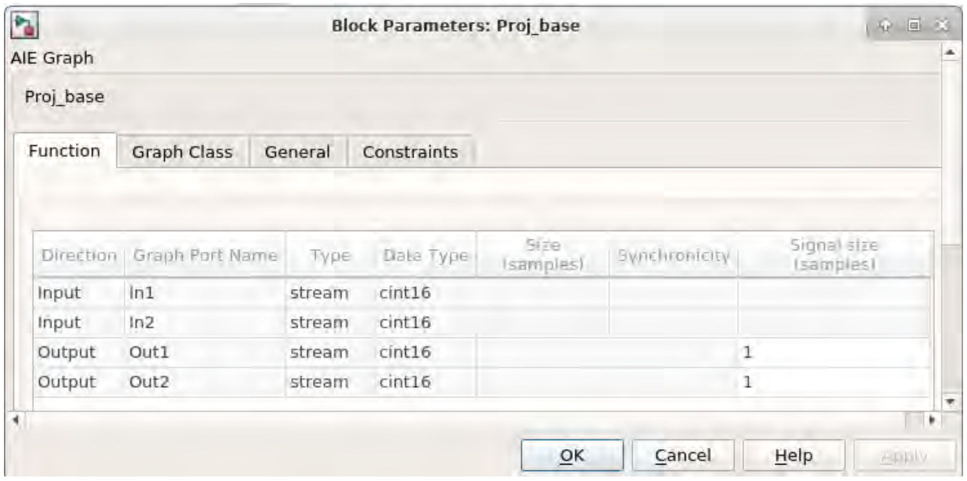


Select runtime parameter (RTP) ports, if any, and click on 'Build' at the bottom.

Direction	Name	RTP
input	In1	☐
input	In2	☐
output	Out1	☐
output	Out2	☐

Block Parameters window opens as shown in the following figure. This contains the port direction and type.

Figure 244: **Block Parameters**


Block Parameters: Proj_base

AIE Graph

Proj_base

Direction	Graph Port Name	Type	Data Type	Size (samples)	Synchronicity	Signal size (samples)
Input	In1	stream	cint16			
Input	In2	stream	cint16			
Output	Out1	stream	cint16			1
Output	Out2	stream	cint16			1

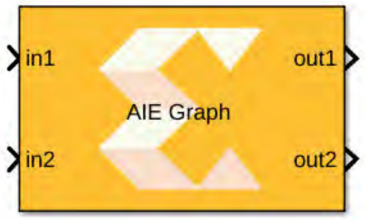
Parameters such as the Graph Port Name, Data Type etc. are automatically updated from the graph code. The only user-editable function configuration parameter is the signal size.

Table 21: **User Editable Parameter**

User-Editable Parameter	Criticality	Description
Signal size	Mandatory	This parameter represents the size of the output signal and should be set to a value equal to or greater than the number of samples that are produced at every invocation of the kernel.

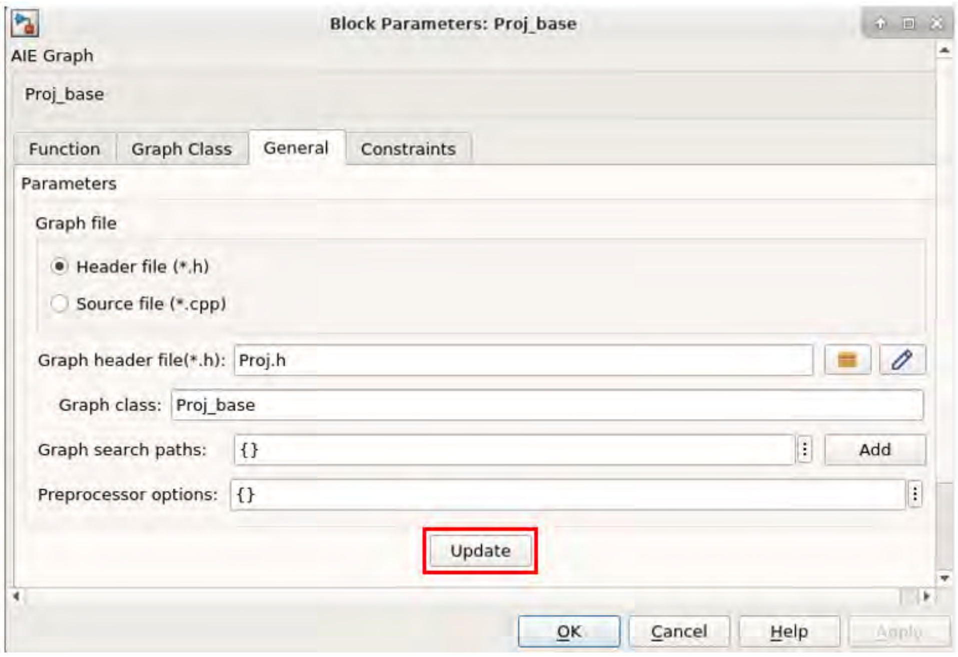
The AI Engine graph block GUI interface with input and output ports is as shown in the following figure.

Figure 245: **AIE Graph Block**



In the General tab, the Import button changes to Update, enabling further update of block parameters.

Figure 246: **Update**


Block Parameters: Proj_base

AIE Graph

Proj_base

Function | Graph Class | General | Constraints

Parameters

Graph file

(● Header file (*.h)) ○ Source file (*.cpp)

Graph header file(*.h): Proj.h

Graph class: Proj_base

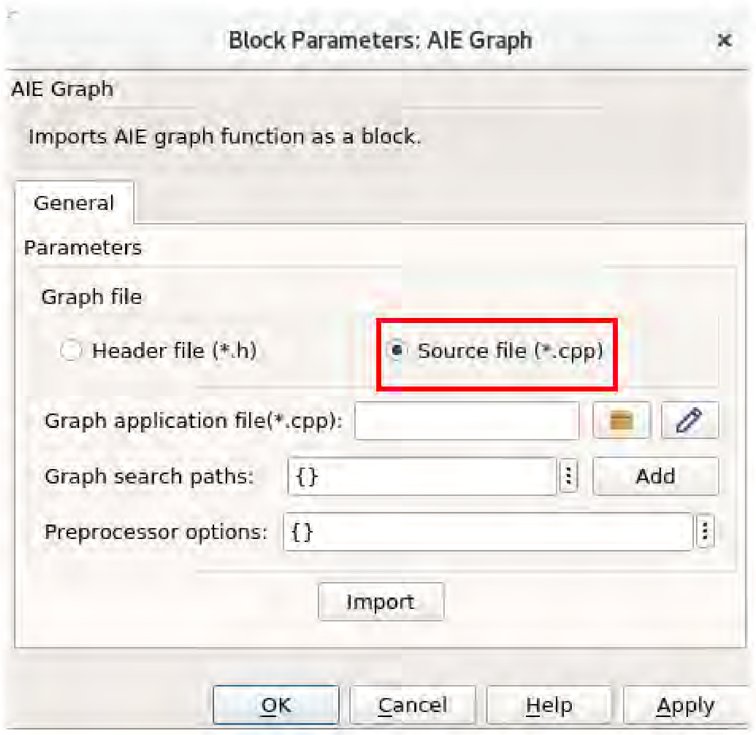
Graph search paths: {}

Preprocessor options: {}

Using the Source File (*.cpp)

To import the graph using the Source file (`.cpp`), double-click the **AIE Graph** block and select the **Source file (*.cpp)** option in the General tab. Specify the graph application file (`*.cpp`), search paths, and preprocessor options as shown in the following figure.

Figure 247: **AI Engine Graph Block Parameters**


Block Parameters: AIE Graph

AIE Graph

Imports AIE graph function as a block.

General

Parameters

Graph file

Header file (*.h) [✓ Source file (*.cpp)]

Graph application file(*.cpp):

Graph search paths: {}

Preprocessor options: {}

The following table provides further details including names and descriptions of each parameter.

Table 22: **AI Engine Graph Block Parameters**

Parameter Name	Parameter Type	Criticality	Description
Graph Application file	String	Mandatory	Specify the file(`.cpp`), where the adf dataflow graph is instantiated and connected to simulation platform. This file should contain the main() function, from where the dataflow graph initializes and runs.
Graph Search paths	Vector of strings	Mandatory	Specify search paths where header files, kernels, and other include files can be found and included for simulation. The search path `$XILINX_VITIS/adf/include` (where `adf.h` isdefined) is be included by default and does not need to be specified.
Preprocessor options		Optional	Optional preprocessor arguments for downstream compilation with specific preprocessor options. The following preprocessor option formats are accepted and multiple can be selected: '`-Dname`' and '`-Dname=definition`'. That is, the optional argument must begin with the '`-D`' string and if the option definition value is not provided, it is assumed to be 1.

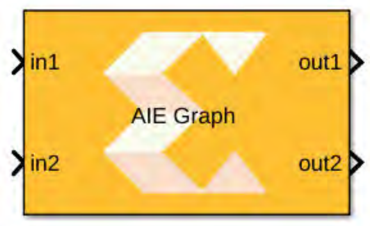
The following example shows the sample graph `.cpp` file. The graph is connected as follows. This file should be pointed to the Graph application file field in the Block parameters GUI.

`graph.cpp`

```
#include "Proj.h"

// instantiate cardano dataflow graph
Proj mygraph;

// initialize and run the dataflow graph
#if defined(__AIESIM__) || defined(__X86SIM__)
int main(void) {
   mygraph.init();
   mygraph.run();
   mygraph.end();
   return 0;
}
#endif
```

When the GUI parameters are updated, click **Import**. The tool generates the graph database and after successful import the AI Engine graph block gets updated with input and output ports as shown in the following figure.

Figure 248: **AIE Graph Block**



Notice that the AIE Graph block interface generated when importing the header file and the source file remains the same.

The Function Tab in the AIE Graph block parameters GUI appears similar to the one shown in the Import graph using the header file. You can update the signal size parameter and click **OK** to exit the Block parameters window.

IMPORTANT!

- To connect an AI Engine graph `inout` port to an AI Engine graph `input` RTP port, the synchronocities of both ports must be compatible, otherwise, an appropriate error is reported by Model Composer.
- If the RTP port's behavior is different from its default behavior, the connection should appropriately specify it as an `async` or `sync` port in graph code.

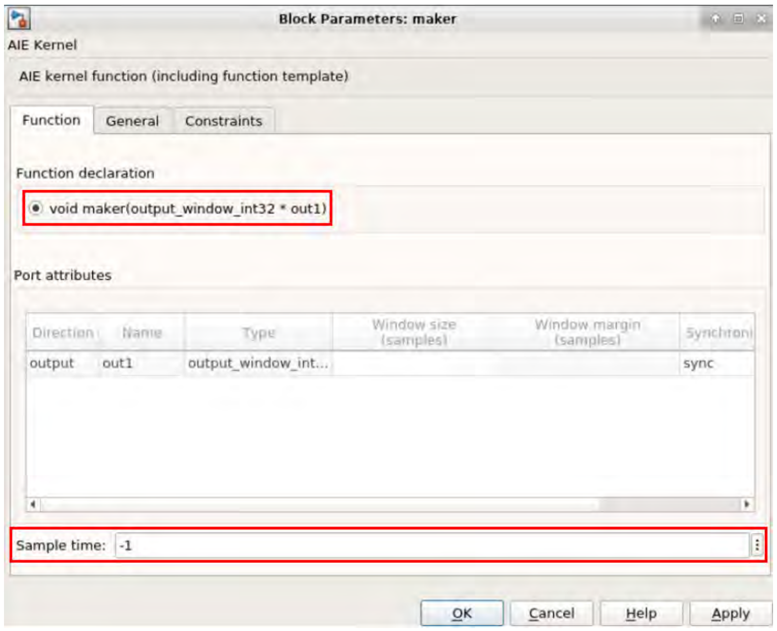
Importing Kernels and Graphs as Source Blocks

Model Composer supports importing kernels and graphs as a source blocks (i.e., user-defined functions with no input ports) into the design. These source blocks have inherited sample time and work only when the model has at least one another block with a non-inherited sample time that Simulink can use for sample time propagation. This requires you to add a block (for example a Constant block) in the model and specify a valid sample time. Alternatively, you can explicitly specify the sample time for the imported source blocks similarly to the other source blocks such as Constant, RTP Source blocks etc. from Simulink and AI Engine libraries respectively.

The AIE Kernel, AIE Class Kernel and AIE Graph blocks from the AI Engine/User-Defined Functions library supports importing the code as a source block. When you try to import the kernel into the Model Composer that has only output ports, the tool identifies that as a source block and adds the 'Sample time' parameter to the Function tab in Block parameters GUI as shown in the following figure.

Note: The Sample time parameter is not visible when the kernel code has at least one input port.

Figure 249: **Block Parameters: maker**



Block Parameters: maker

AIE Kernel

AIE kernel function (including function template)

Function | General | Constraints

Function declaration

● void maker(output_window_int32 * out1)

Port attributes

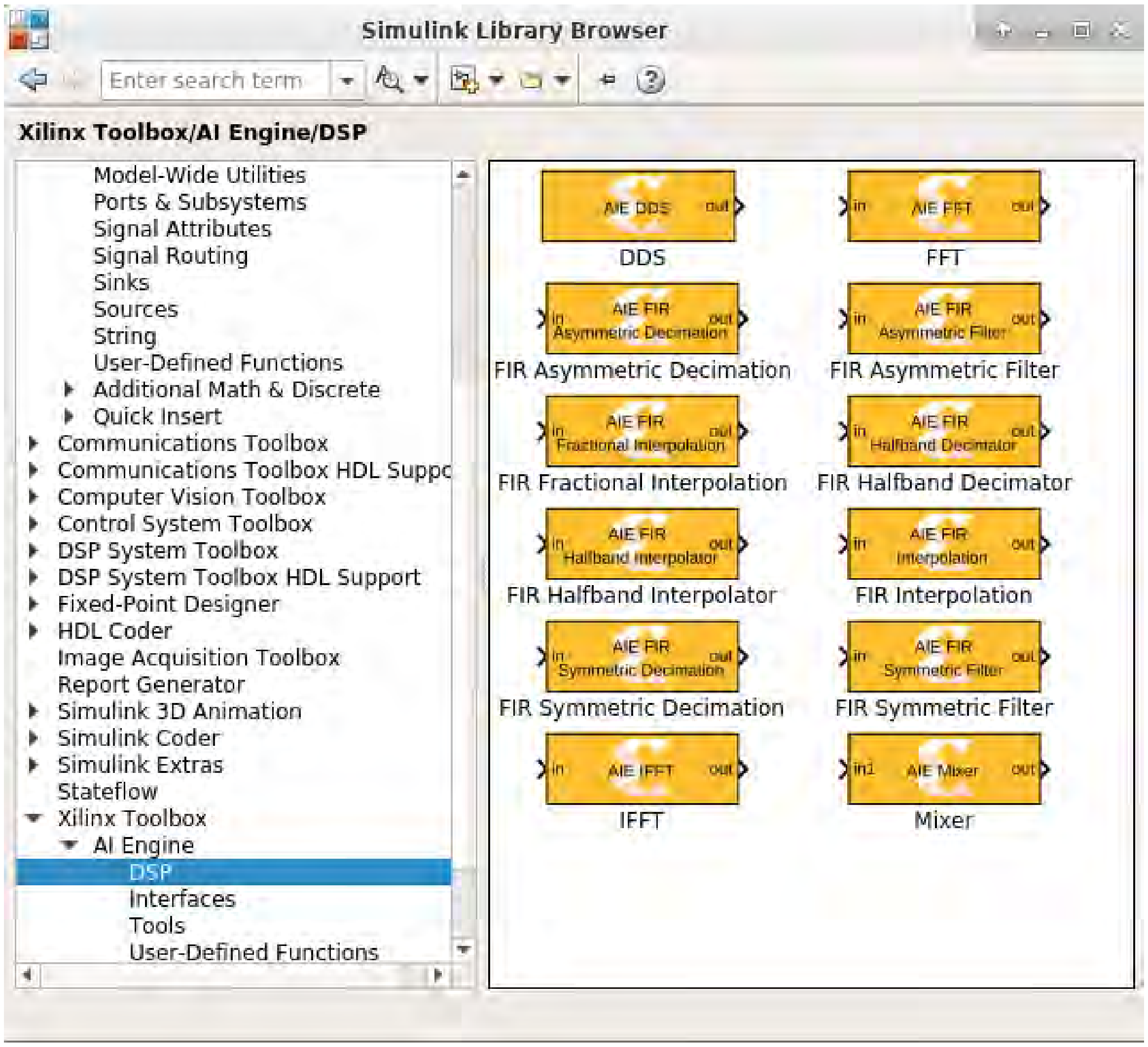
Direction	Name	Type	Window size (samples)	Window margin (samples)	Synchroni
output	out1	output_window_int...			sync

Sample time: -1

The default value of the Sample time is `-1` which indicates the sample time is inherited. Valid sample time values must be positive, real scalar, or -1.

AI Engine DSPLib

The Vitis™ DSP Library (DSPLib) is a library of commonly used DSP functions optimized for AI Engines. To facilitate the use of these functions in a design, Vitis Model Composer provides different DSPlib functions as blocks within the Xilinx Toolbox/AI Engine/DSP library. You can conveniently drag and drop one of these blocks into your model from the Simulink Library browser and configure the block.



Vitis Model Composer provides a copy of DSPLib functions which can be used as is. Optionally, you can download DSPLib functions from the online GitHub repository and use the MATLAB command `xmcLibraryPath` to set the path to the DSPLib library.

For more information on using the MATLAB utility, refer to xmcLibraryPath.

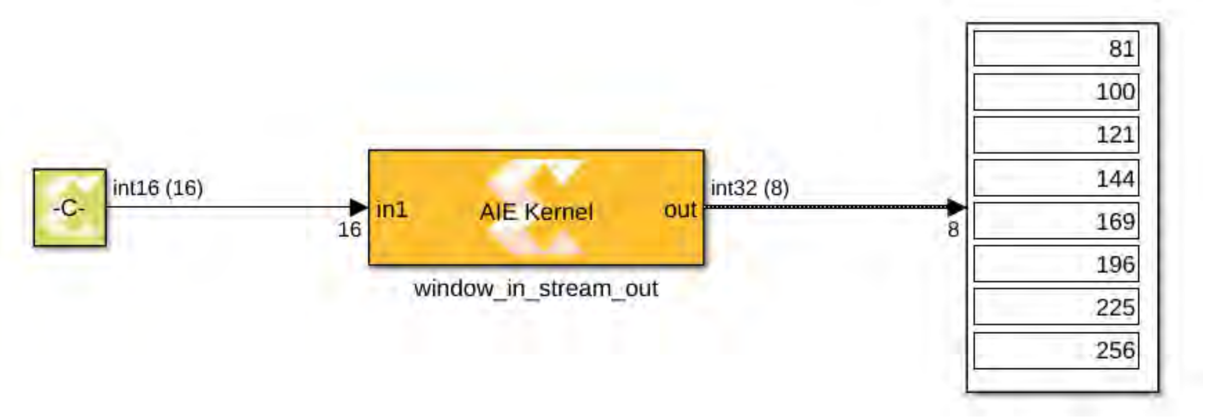
Setting Signal Size to Avoid Buffer Overflow

The Signal Size field on the AI Engine import block masks only applies to kernels with stream or cascade outputs. Moreover, it has no implementation significance and it is only meaningful for simulation purposes in the Simulink environment. This section provides more in-depth knowledge of what Signal Size is and how to set it.

Start with a very simple kernel with window input and stream output. The kernel code is as follows:

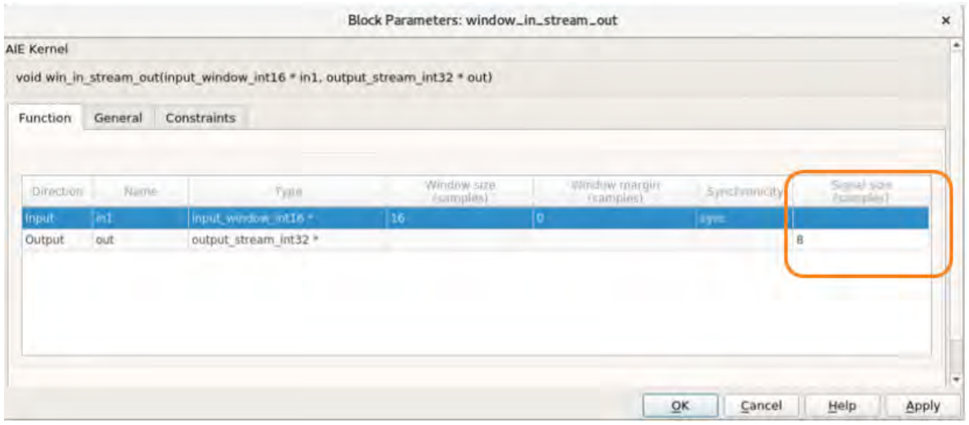
```
void win_in_stream_out(input_window_int16 * in1,output_stream_int32 * out) {
  int16 val;
  for (unsigned i=0; i<16; i++) {
    window_readincr(in1,val);
    int32 squaring = val * val;
    writeincr(out,squaring);
  }
}
```

Figure 250: **AIE Kernel: Window Input/Stream Output**



This kernel expects a window of size 16 and at every invocation of this kernel, 16 output samples are generated. Import this kernel into Simulink using the AIE Kernel block. The mask for the block is shown in the following figure.

Figure 251: **Block Parameters: window_in_stream_out**


Block Parameters: window_in_stream_out

AIE Kernel

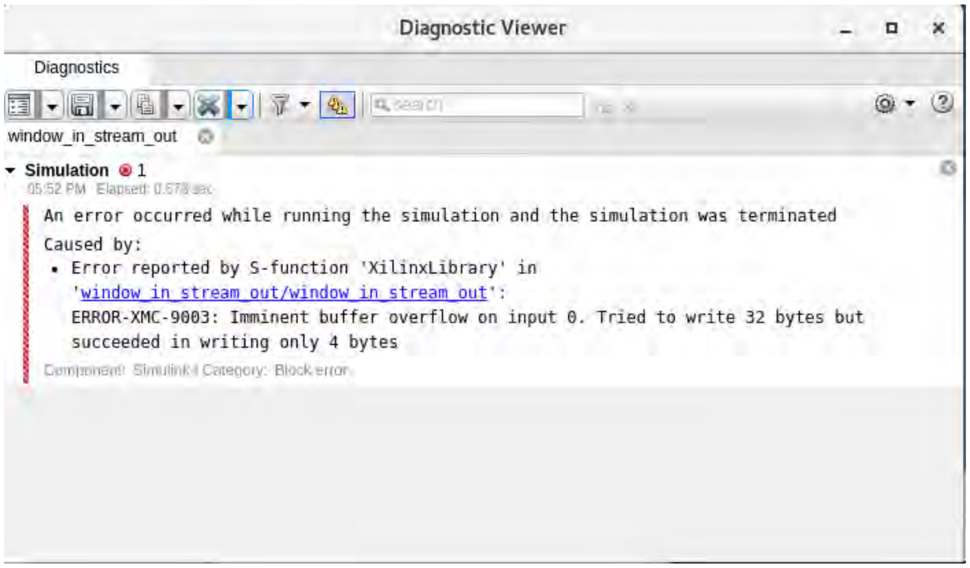
void win_in_stream_out(input_window_int16 * in1, output_stream_int32 * out)

Direction	Name	Type	Window size (samples)	Window margin (samples)	Synchronicity	Signal size (samples)
Input	in1	input_window_int16 *	16	0	sync	
Output	out	output_stream_int32 *				8

Regardless of what value you set the signal size to, it does not affect the numerical output. For this example, you will generally set the signal size to 16 because every invocation of the kernel produces 16 samples. In this case, the output of this block will be a variable size signal of maximum size 16 (equal to the signal size) and each output will contain 16 samples. However, if for example you set the signal size to 32, the output of the block will be a variable size signal with a maximum size of 32, but each output will only contain 16 samples.

What if you set the signal size to a number smaller than 16, for example to 8? In this case, similar to the previous cases, the output will be a variable size signal of maximum size of 8. As mentioned previously, at each invocation of the kernel, the kernel produces 16 samples. Eight of these samples will be put out by the block. The other eight are stored in an internal buffer in the block. If you call the kernel too many times, eventually the internal buffer of the block will fill up and you will see a buffer overflow error as shown in the following figure.

Figure 252: **Buffer Overflow Error**

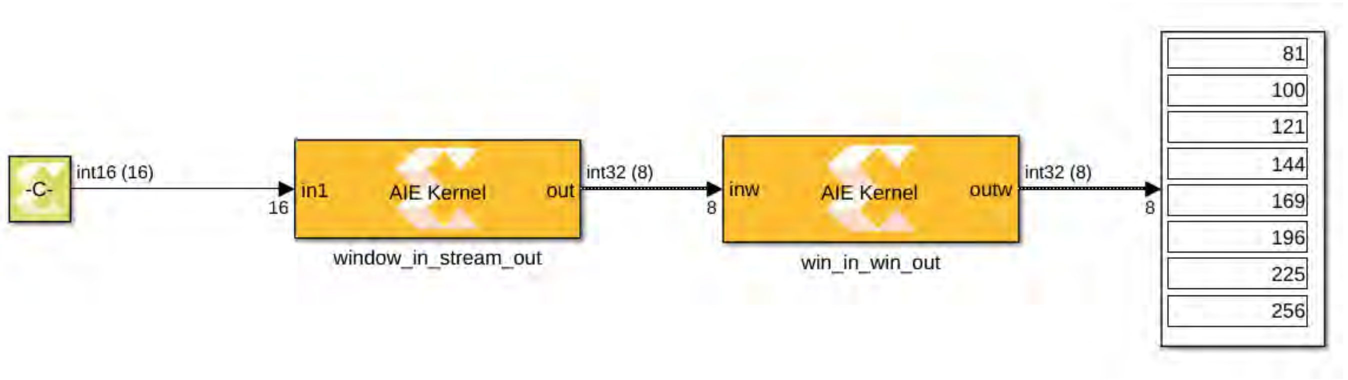


```
An error occurred while running the simulation and the simulation was terminated
Caused by:
 • Error reported by S-function 'XilinxLibrary' in
   'window_in_stream_out/window_in_stream_out':
   ERROR-XMC-9003: Imminent buffer overflow on input 0. Tried to write 32 bytes but
   succeeded in writing only 4 bytes
Component: Simulink | Category: Block error
```

This is a trivial example. You may contend that there is no reason to set the signal size to anything less than 16, and that is correct. Now examine a model with two AI Engine kernels. Connect the output of the kernel previously created to another AI Engine kernel with window input and window output. The code for this second kernel is as follows:

```
void win_in_win_out(input_window_int16 * inw, output_window_int16 * outw)
{
    int16 temp;
    for (unsigned i=0; i<8; i++) {
        window_readincr(inw,temp);
        window_writeincr(outw,temp);
    }
}
```

Figure 253: **Two AIE Kernels**



This kernel requires an input window of size 8 and produces a window size of 8. Now consider two scenarios. First consider a case in which the first block has the signal size set to 16. As mentioned previously, with a signal size of 16, the buffer for the first block will not overflow. But now examine the second block more closely. The second kernel upon receiving 16 samples, will get invoked twice. Each time, it produces eight samples for a total of 16 samples. However, since the output window size is 8, the block will produce eight samples and store the other eight in the internal buffer. Just like before, if we run this model for long enough, the buffer for the second block will overflow and simulation will stop.

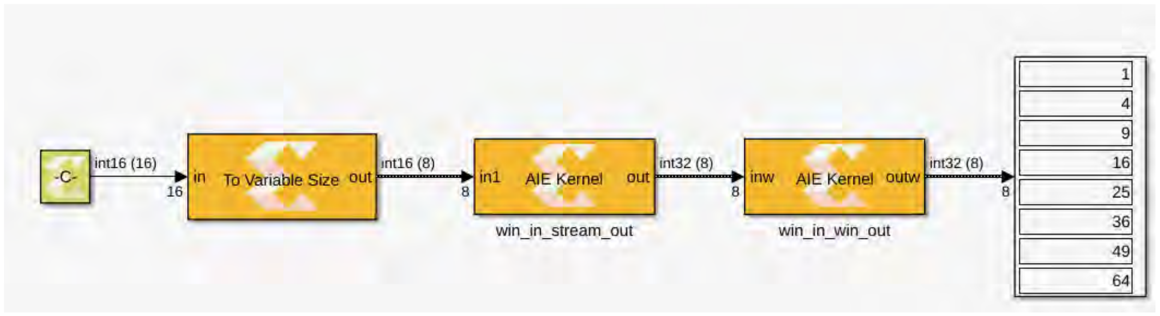
In another scenario, to avoid an overflow, we might set the signal size for the first block to 8. This will avoid an overflow in the second block. However as mentioned previously, now the buffer for the first block will overflow. So how can we get out of this situation?

The buffer overflows because we are feeding more data to the blocks than the blocks can process. If we reduce the rate, the kernels will be able to process any excess data in the buffers and as such prevent the overflow. Now look into this more carefully.

Assume the simulation has been running for a while and the first block's buffer is not empty. If we somehow stop feeding data to the first block, every time simulink calls the first block, the kernel will not be invoked (there is no input data), but because there are samples in the buffer, the block will continue to produce samples (eight at a time) until the buffer empties out after which it will produce an empty variable size signal.

This information should help you avoid buffer overflow. Instead of stopping the input as suggested above, simply reduce the flow of the data into the first block. One way of doing this is to use a To Variable Size block from AI Engine/Tools and set the Output size on the block mask to a number smaller than the size of the input. The following figure depicts the same design shown above but with a To Variable Size block at its input.

Figure 254: **Two AIE Kernels: Buffer Block Input**



In this design, because fewer samples are being fed to the first block at any given call to the block, the buffers will not overflow. Note that the output of this model will be different from the model without the buffer block because the buffer block produces zero samples at time step zero.

TIP: *If a block with stream output is connected to a block with window input, set the size of the signal size for the producing block to the same size as the input window for the consuming block.*

TIP: *To avoid buffer overflow, reduce the rate you feed data to the system using a buffer block.*

Simulation and Code Generation

After a high level graphical design is created using the blocks available in the Vitis Model Composer AI Engine library, it should be simulated interactively in the Simulink environment. This process ensures the functional correctness of the design using the native Simulink functional simulator and displays the results on scopes and graphical displays. The compilation and execution times are generally short at this stage, which helps you to quickly verify the functionality and iterate over the design until the specification requirements are met. The functionally verified design can then be used to generate the dataflow graph using the Model Composer Hub block. The verification of the dataflow graph can be done using various execution targets which Model Composer supports to simulate your AI Engine application at different levels of abstraction, accuracy, and speed.

This section discusses following topics in detail:

- Running Simulink Simulation
- Code Generation
- Verifying the generated dataflow graph

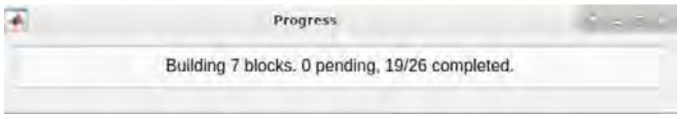
Running Simulink Simulation

Simulation involves compiling the design and checking for any design rule violations, then executing the design to produce the outputs. You can define the inputs of the design using any Simulink source blocks and the output is analyzed either by logging the data to the workspace, or by visually viewing the results using a scope, spectrum analyzer, or display block.

Model Composer provides two MATLAB utilities `xmcVitisRead` and `xmcVitisWrite` to directly read/write data from/to the files that are formatted for AIE Simulator and/or x86 Simulator. For more information on using these utilities, refer to AI Engine Utilities.

Clicking the **Simulate model** icon in the Simulink simulation tool bar, compiles all the kernels and graphs in the design. You can monitor the status from the Progress window, which displays after simulation begins (see the following figure).

Figure 255: **Compilation Status**



Building 7 blocks. 0 pending, 19/26 completed.

The Compilation Status window displays only when you are compiling a design for the first time. When you simulate the model again, unless you make any changes to the imported kernel or graph, Model Composer will use the cached entry for the block to run the simulation faster. For example, assume you have three kernels (`add2`, `add3`, `add4`) in your design and you run the simulation for the first time. In that case, all the three kernels get compiled. When you change the `add3` kernel code and try to simulate again, only the changed `add3` kernel gets re-compiled and the cached entries for `add2` and `add4` are used for faster simulation.

To manage the simulation cache in Model Composer, use the commands described in Managing the HLS Block Cache from the MATLAB command prompt.

When simulation is complete, you can review the results by connecting any of the Simulink sink blocks to appropriate points in your design.

Code Generation

When the design is functionally verified in Simulink, you can generate the dataflow graph from the design. It is necessary to encapsulate the AI Engine blocks into a subsystem. To understand more about creating a top-level subsystem, refer to Creating a Top-Level Subsystem Module.

IMPORTANT! *To generate output from the AI Engine model, only blocks from the Vitis Model Composer AI Engine library and a limited set of Simulink blocks can be used in the subsystem that is instantiated at the top-level of the design. Refer to Connecting Source and Sink Blocks for more details about the blocks that are supported inside the subsystem.*

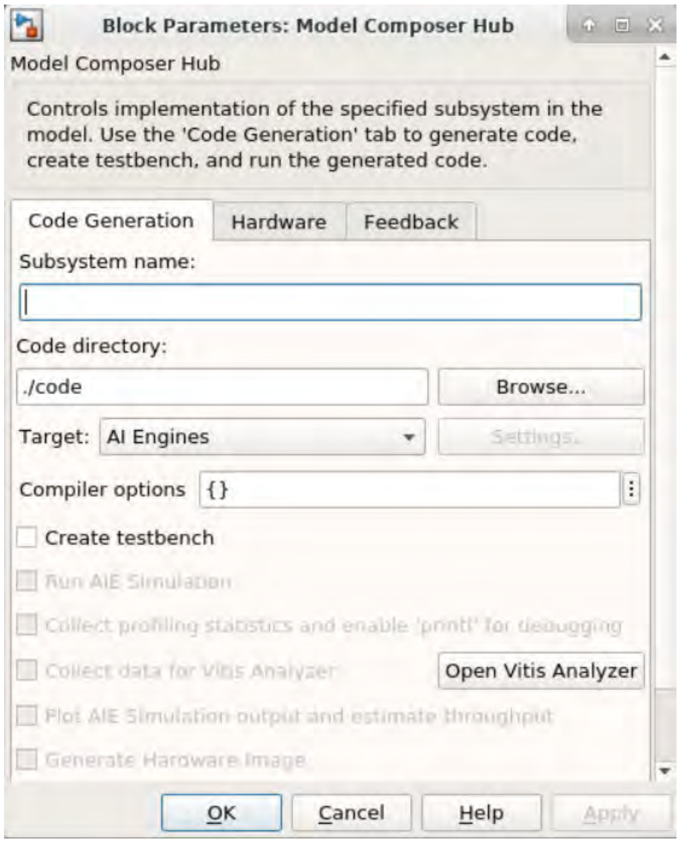
Model Composer Hub Block for AI Engine Code Generation

Model Composer automatically generates AI Engine code (dataflow graph) from the subsystem that comprises blocks from the AI Engine Blockset library. However, an AI Engine model in Model Composer requires the addition of the Model Composer Hub block to configure compilation and generation of AI Engine output. In addition to the targets available in the Model Composer Hub block which supports compilation of the design into low-level representations using blocks from the library, it also supports an AI Engine compilation target.

This section discusses only the graph code generation from Model Composer. For running and verifying the generated AI Engines code, refer to Verification of AI Engine Code.

The Model Composer Hub block and the block parameters dialog box specific to AI Engine compilation targets are shown in the following figure.

Figure 256: **Model Composer Hub and Parameters**


Block Parameters: Model Composer Hub

Model Composer Hub

Controls implementation of the specified subsystem in the model. Use the 'Code Generation' tab to generate code, create testbench, and run the generated code.

Subsystem name:

Code directory: ./code

Target: AI Engines

Compiler options {}

Create testbench

Run AIE Simulation

Collect profiling statistics and enable 'printf' for debugging

Collect data for Vitis Analyzer

Plot AIE Simulation output and estimate throughput

Generate Hardware Image

When you add the Model Composer Hub block from the library, the Target is set to **AI Engines** by default as shown in the previous figure. For more details about adding the Model Composer Hub block into the design and associated features, refer to Adding the Model Composer Hub. The **Subsystem name** field should be given the top-level subsystem module name.

You can specify a cell array of AI Engine compiler options using the **Compiler Options** edit button. This provides a method to control the compiler debug options, execution target options, file options and so on. Examples as follows:

- To control the debug option log-levels, you can specify the string `{'--log-level=5'}` in the **Compiler Options** field.
- For issues related to the stack size or heap size in the downstream AI Engine flows, you can increase the size by adding `--stacksize=<int>` and `--heapsize=<int>` in the Compiler options field.

When the code directory, subsystem name, and target are specified, click **Generate** to create the dataflow graph.

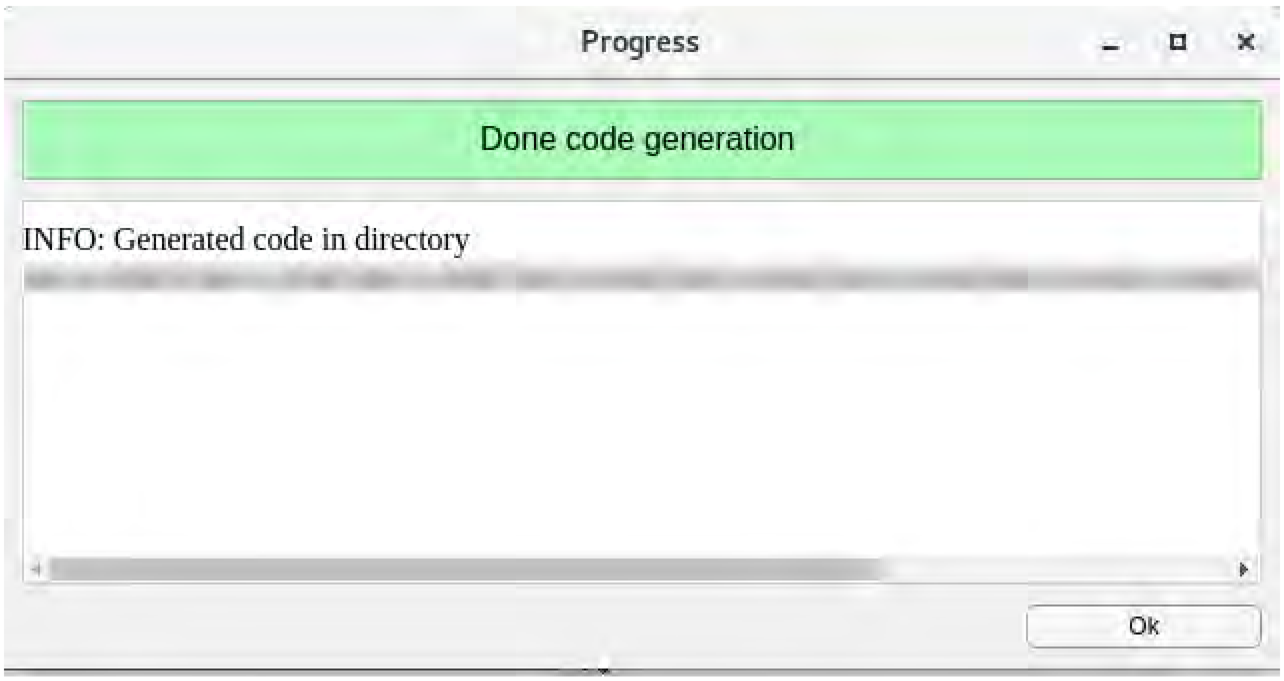
IMPORTANT! *You can enable the **Create testbench** option to log the test data at input and output. Further, you can enable the **Run AIE Simulation** option to verify the dataflow graph. For more information, refer to Verification of AI Engine Code.*

Code generation begins when you click **Apply** to confirm any changes and then click **Generate**.

Once the code generation process is initiated, the Compilation Status window may display (based on whether the imported kernel/graph code is pre-compiled or not). If the design was already compiled during the Simulink simulation phase, you may not see this window. However, Model Composer displays the progress of code generation in the Progress window.

When Model Composer has completed generating the code, it displays the status message **Done code generation** in the Progress window as shown in the following figure.

Figure 257: **Done Code Generation**



```
INFO: Generated code in directory
```

TIP: *Model Composer runs the Simulink simulation every time you try to generate the code. You can choose to either run the simulation manually or click **Generate** which validates the subsystem by running a set of DRCs. If the DRC validation fails, then Model Composer returns an appropriate error and the code generation process stops.*

Output Directory

A new directory gets created with the name specified in the `code directory` field in the Model Composer Hub block. There are various sub-directories in the `code` directory but the `src_aie/` directory and the `Makefile` which are highlighted in figure below, are of interest for this section. The details about other sub-directories are explained in subsequent topics.

Figure 258: **Target Directory**

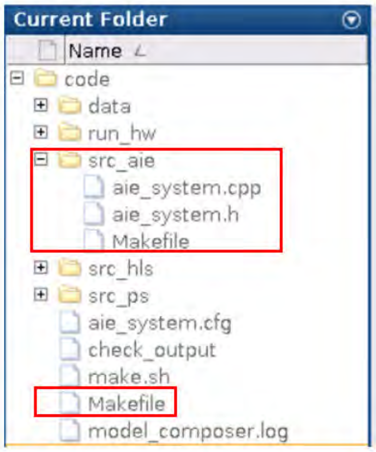
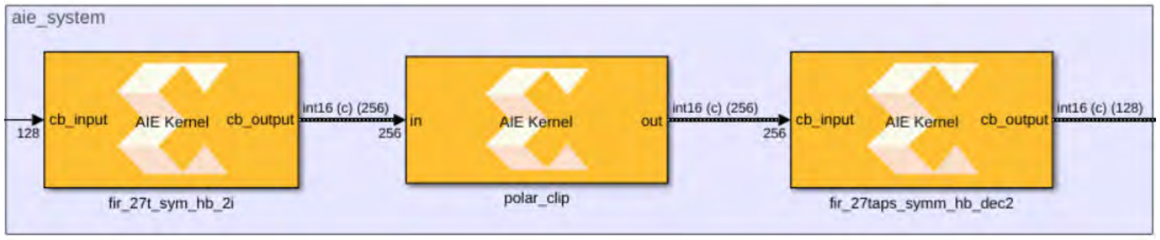


Table 23: **File/Directory Descriptions**

File/Directory	File/Sub-directory	Description
`src_aie`	`Subsystem_Name.h`	Header file that specifies the AI Engine dataflow graph ***Note***: `Subsystem_Name` is a unique string derived from the top-level subsystem specified in the Model Composer model.
	`Subsystem_Name.cpp`	Test bench to simulate the dataflow graph specified in `Subsystem_Name.h`.
	Makefile	This file contains the `aiecompiler` options specified from Model Composer Hub block
Makefile		This file contains the commands to compile and simulate the dataflow graph. For more details on how to use the `Makefile` to run simulation from command line, refer to the section Running Simulation using the Makefile.

The files generated by Model Composer in the `src_aie` directory reflect the contents and hierarchy of the subsystem that gets compiled. In this case, assume the subsystem is `aie_system` which is derived from the design in *Vitis Model Composer Tutorial* (UG1498). The following figure shows the interconnection of imported kernel functions as blocks, encapsulated as a subsystem.

Figure 259: **Block Interconnection**



The generated code for the subsystem `aie_system` is as follows.

`aie_system.h`

```
      #ifndef __XMC_AIE_SYSTEM_H__
#define __XMC_AIE_SYSTEM_H__

#include <adf.h>
#include "kernels/inc/hb_27_2i.h"
#include "kernels/inc/polar_clip.h"
#include "kernels/inc/hb_27_2d.h"

class Aie_system_base : public adf::graph {
public:
   adf::kernel fir_27t_sym_hb_2i_0;
   adf::kernel polar_clip_0;
   adf::kernel fir_27taps_symm_hb_dec2_0;

public:
   adf::input_port In1;
   adf::output_port Out1;

   Aie_system_base() {
      // create kernel fir_27t_sym_hb_2i_0
      fir_27t_sym_hb_2i_0 = adf::kernel::create(fir_27t_sym_hb_2i);
      adf::source(fir_27t_sym_hb_2i_0) = "kernels/src/hb_27_2i.cpp";

      // create kernel polar_clip_0
      polar_clip_0 = adf::kernel::create(polar_clip);
      adf::source(polar_clip_0) = "kernels/src/polar_clip.cpp";

      // create kernel fir_27taps_symm_hb_dec2_0
      fir_27taps_symm_hb_dec2_0 =
adf::kernel::create(fir_27taps_symm_hb_dec2);
      adf::source(fir_27taps_symm_hb_dec2_0) = "kernels/src/hb_27_2d.cpp";

      // create kernel constraints fir_27t_sym_hb_2i_0
      adf::runtime<ratio>( fir_27t_sym_hb_2i_0 ) = 0.9;

      // create kernel constraints polar_clip_0
      adf::runtime<ratio>( polar_clip_0 ) = 0.9;

      // create kernel constraints fir_27taps_symm_hb_dec2_0
      adf::runtime<ratio>( fir_27taps_symm_hb_dec2_0 ) = 0.9;

      // create nets to specify connections
      adf::connect< adf::window<512,64> > net0 (In1,
fir_27t_sym_hb_2i_0.in[0]);
      adf::connect< adf::window<1024>, adf::stream > net1
(fir_27t_sym_hb_2i_0.out[0], polar_clip_0.in[0]);
      adf::connect< adf::stream, adf::window<1024,128> > net2
(polar_clip_0.out[0], fir_27taps_symm_hb_dec2_0.in[0]);
      adf::connect< adf::window<512> > net3
(fir_27taps_symm_hb_dec2_0.out[0], Out1);
   }
};

class Aie_system : public adf::graph {
public:
   Aie_system_base mygraph;
public:
   adf::input_plio In1;
   adf::output_plio Out1;

   Aie_system() {
      In1 = adf::input_plio::create("In1",
            adf::plio_32_bits,
            "./data/input/In1.txt");

      Out1 = adf::output_plio::create("Out1",
            adf::plio_32_bits,
            "Out1.txt");

      adf::connect< > (In1.out[0], mygraph.In1);
      adf::connect< > (mygraph.Out1, Out1.in[0]);
   }
};

#endif // __XMC_AIE_SYSTEM_H__
```

Model composer automatically generates the graph header file `aie_system.h` which contains the dataflow graph corresponding to the subsystem name specified in the Model Composer Hub block. The connection between the AI Engine kernel or graph as well as the configuration parameters such as window size, window margin and so on, which are specified in the AI Engine kernel block, are automatically reflected in the generated graph code.

Model Composer also generates the `aie_system.cpp` file, which is a control program that has the `main()` function defined to initialize, run, and end the simulation using the control APIs.

`aie_system.cpp`

```
#include "aie_system.h"

// instantiate cardano dataflow graph
Aie_system mygraph;

// initialize and run the dataflow graph
#if defined(__AIESIM__) || defined(__X86SIM__)
int main(void) {
   mygraph.init();
   mygraph.run();
   mygraph.end();
   return 0;
}
#endif
```

Limitations

- AI Engine code cannot be generated if a top-level subsystem's `output` port is connected to a synchronous run-time parameter port.
- The value of an input run-time parameter port can only be updated once at the beginning when running the generated graph.

Stream FIFO Depth Specification

The AI Engine architecture uses streaming data extensively for communicating between two AI Engines, and for communicating between the AI Engine and the programmable logic (PL). This raises the potential for a resource deadlock when the data flow graph has reconvergent stream paths. If the pipeline depth of one path is longer than the other, the producer kernel can stall and might not be able to push data into the shorter path because of back pressure. At the same time, the consumer kernel is waiting to receive data on the longer path due to the lack of data. If the order of data production and consumption between two stream paths is different, a deadlock can happen even between two kernels that are directly connected with two stream paths.

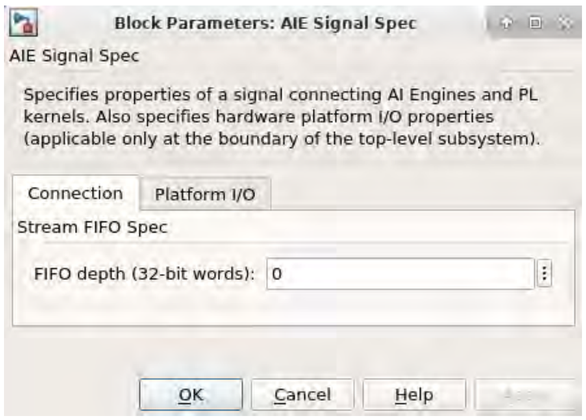
Model Composer supports adding a FIFO_DEPTH between two AI Engine kernels or between an AI Engine and the programmable logic (PL) using the AIE Signal Spec block and automatically generates the graph code with `fifo_depth` constraint on a connection.

Figure 260: **AIE Signal Spec**

AIE Signal Spec
AIE Signal Spec

The AIE Signal Spec block specifies properties of a signal connecting AI Engines and PL kernels. When you double click the AIE Signal Spec block, notice that the block parameters consists of two tabs: Connection and PlatformI/O.

Figure 261: **Block Parameters: AIE Signal Spec**


Block Parameters: AIE Signal Spec

AIE Signal Spec

Specifies properties of a signal connecting AI Engines and PL kernels. Also specifies hardware platform I/O properties (applicable only at the boundary of the top-level subsystem).

Stream FIFO Spec

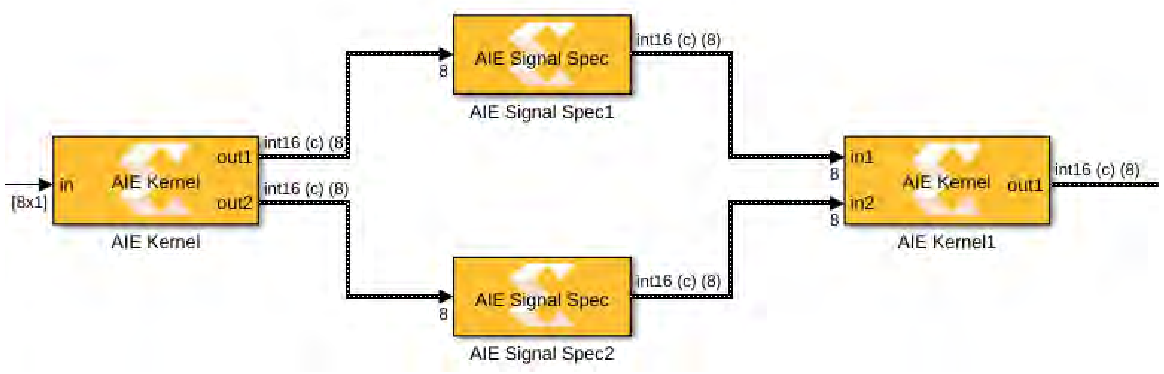
FIFO depth (32-bit words): 0

- **Connection:** Use this tab to specify FIFO depth

- **PlatformI/O:** Use this tab to specify platform I/O properties (for more details on this topic, refer to PLIO Attributes).

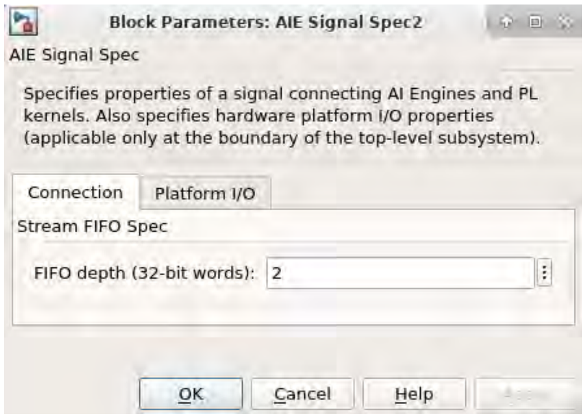
Consider the following example where the AIE Signal Spec blocks are connected in two stream paths between AI Engine kernel blocks.

Figure 262: **AIE Signal Blocks in Two Stream Paths**



The FIFO depth value is are specified as `0` by default and the corresponding information is reflected on the block symbol. For example, if values `2` and `4` are specified as parameters of the two AIE Signal Spec blocks, the GUI will update as shown in the following figures.

Figure 263: **AIE Signal Spec1**


Block Parameters: AIE Signal Spec2

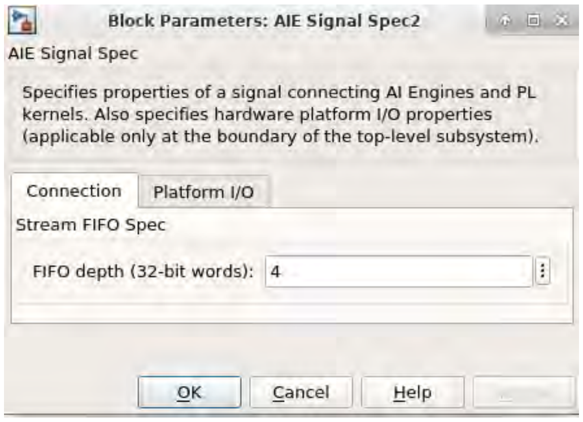
AIE Signal Spec

Specifies properties of a signal connecting AI Engines and PL kernels. Also specifies hardware platform I/O properties (applicable only at the boundary of the top-level subsystem).

Stream FIFO Spec

FIFO depth (32-bit words): 2

Figure 264: **AIE Signal Spec2**



Block Parameters: AIE Signal Spec2

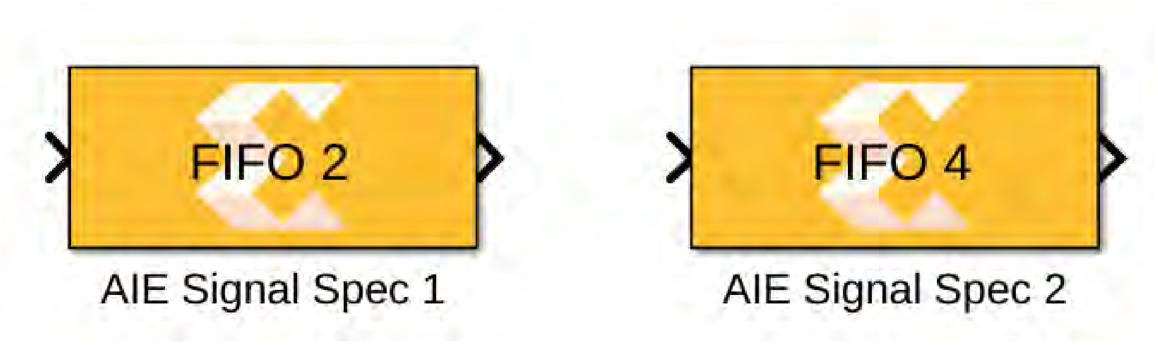
AIE Signal Spec

Specifies properties of a signal connecting AI Engines and PL kernels. Also specifies hardware platform I/O properties (applicable only at the boundary of the top-level subsystem).

Stream FIFO Spec

FIFO depth (32-bit words): 4

Figure 265: **Block GUI Updates**



The stream FIFO values specified using the AIE Signal Spec block are automatically updated in the generated graph code (`graph.h`) with `fifo_depth` constraints as shown in the following code.

Snippet of `graph.h`

```
// create nets to specify connections
        adf::connect< adf::stream > net0 (In1, AIE_Kernel.in[0]);
        adf::connect< adf::stream > net1 (AIE_Kernel.out[0],
AIE_Kernel1.in[0]);
        adf::fifo_depth(net1) = 2;
        adf::connect< adf::stream > net2 (AIE_Kernel.out[1],
AIE_Kernel1.in[1]);
        adf::fifo_depth(net2) = 4;
        adf::connect< adf::stream > net3 (AIE_Kernel1.out[0], Out1);
```

Notice the two `fifo_depth` constraints corresponding to the two stream paths in the design. Using this `fifo_depth` constraint on a connection is useful as it creates more buffering in paths with back pressure, and hence avoids any deadlock.

IMPORTANT! *The AIE Signal Spec block can only be used in an AI Engine subsystem.*

PLIO Attributes

Typically, the AI Engine array runs at a higher clock frequency (between 1 GHz and 1.25 GHz) than the internal Programmable Logic. Within the AI Engine core the streaming data-width is 32-bit; whereas between the AI Engine interface tile and PL interface, it is 64-bit by default. To balance the throughput between AI Engine and internal programmable logic, it is desirable to pipeline enough data by choosing wider stream data paths for PL blocks. For example, to utilize an AI Engine array running at 32 bits / 1GHz rate to its full potential, PL blocks may use 64 bits / 500 MHz rate or 128 bits / 250 MHz rate and so on. Such wider (> 32 bits) stream data is sequentialized automatically into 32-bit streams within the AI Engine interface tile.

Specifying PLIOs in Model Composer designs

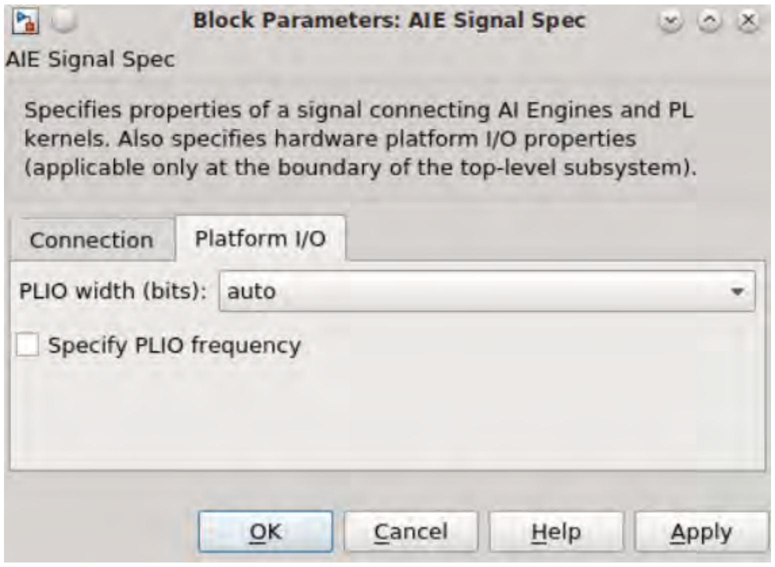
Model Composer supports specifying the PLIO width using the AIE Signal Spec block and making external stream connections that cross the AI Engine to PL boundary.

Figure 266: **AIE Signal Spec**

AIE Signal Spec
AIE Signal Spec

The AIE Signal Spec block supports specifying the hardware platform I/O properties at the boundary of the AI Engine subsystem, along with the FIFO depth information as discussed in Stream FIFO Depth Specification.

Figure 267: **Block Parameters: AIE Signal Spec**



Block Parameters: AIE Signal Spec

AIE Signal Spec

Specifies properties of a signal connecting AI Engines and PL kernels. Also specifies hardware platform I/O properties (applicable only at the boundary of the top-level subsystem).

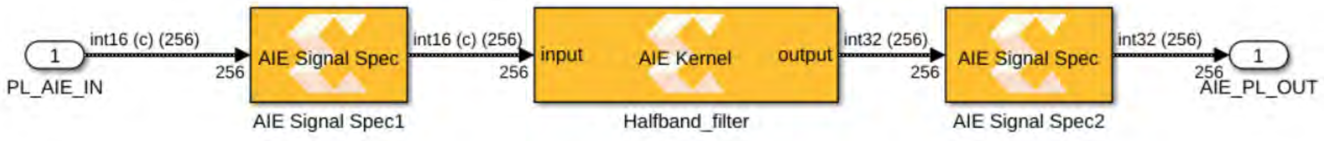
PLIO width (bits): auto

☐ Specify PLIO frequency

From the Platform I/O tab in the AIE Signal Spec block, you can select the available PLIO width options from the drop-down menu.

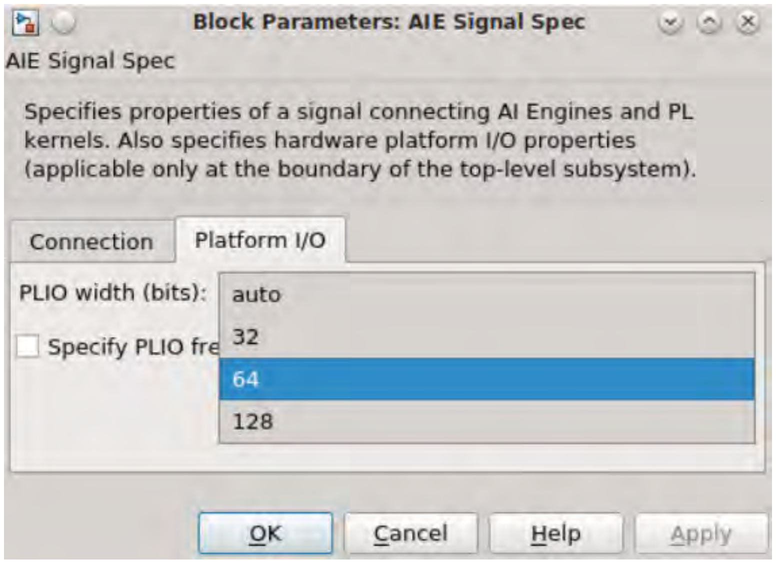
Consider following example where an AIE Signal Spec block is connected at the boundary of the AI Engine subsystem.

Figure 268: **AIE Signal Spec Block Connected to AI Engine Subsystem**



By default, the PLIO width is set to `auto`. Other available options are: `32`, `64`, and `128`.

Figure 269: **Block Parameters: AIE Signal Spec PLIO Width**

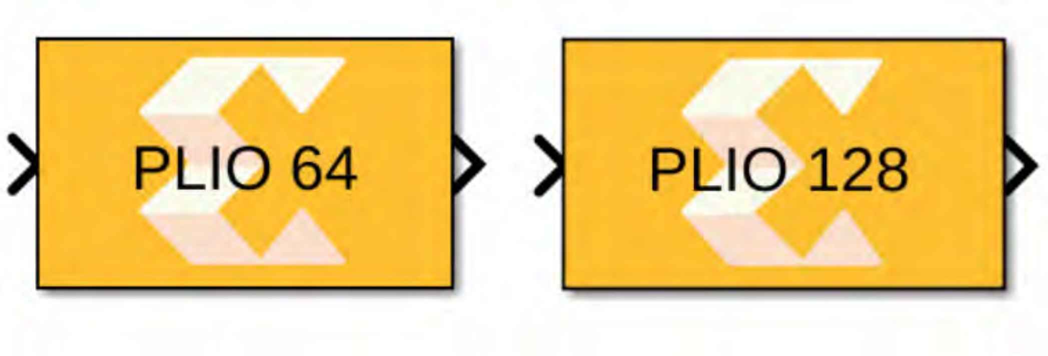

Block Parameters: AIE Signal Spec

AIE Signal Spec

Specifies properties of a signal connecting AI Engines and PL kernels. Also specifies hardware platform I/O properties (applicable only at the boundary of the top-level subsystem).

After the PLIO width is specified as `64` and `128` for the two AIE Signal Spec blocks in the example, the GUI updates as follows.

Figure 270: **AIE Signal Spec Blocks Updated**



The PLIO width specified using the AIE Signal Spec block is automatically updated in the generated graph code (`graph.cpp`) with PLIO constraints as shown in the following code.

Note: Adding the PLIO width does not impact the Simulink simulation, only the code generation.

Snippet of `graph.h`

```
adf::input_plio PL_AIE_IN;
adf::output_plio AIE_PL_OUT;

   Subsystem() {
      PL_AIE_IN = adf::input_plio::create("PL_AIE_IN",
            adf::plio_64_bits,
            "./data/input/PL_AIE_IN.txt");
      AIE_PL_OUT = adf::output_plio::create("AIE_PL_OUT",
            adf::plio_128_bits,
            "AIE_PL_OUT.txt");

 adf::connect< > (PL_AIE_IN.out[0], mygraph.PL_AIE_IN);
 adf::connect< > (mygraph.AIE_PL_OUT, AIE_PL_OUT.in[0]);
```

The PLIO attributes are used in a program to read input from a file or write output data to a file. You can see a connection with one 64-bit PLIO attribute declared for input and one 128-bit PLIO attribute declared for output.

Data File Layout

When simulating PLIO with data files, the data should be organized to accommodate both the width of the PL block as well as the data type of the connecting port on the AI Engine block. Model Composer automatically generates the data file that accommodates to the specified PLIO width.

For example, a data file representing a 64-bit PL interface to an AI Engine kernel expecting `cint16` should be organized as four columns per row, where each column represents a 16-bit real or imaginary value.

PLIO Width	AIE Kernel Data Type	Dat File Layout

64-bit	cint16	`0 0 0 0`
		`1 1 1 1`
		`2 2 2 2`

This data file is in the output code directory.

Specifying PLIO Frequency

The AI Engine always run at 1 GHz and can write (at most) two streams with a 32-bit data width per cycle. In contrast, an IP implemented in the PL can run at up to 500 MHz, while consuming a larger bit-width. In order to balance the throughput between the AI Engine and PL, and also ensure the processes do not create a bottleneck with respect to the total performance, it is required to match the rates between the two. Model Composer supports specifying the frequency of PL from Platform I/O tab in AIE Signal Spec block.

You can either adjust the PLIO frequency or the Width to match the rate between the AI Engine and PL. Consider an example of a 32-bit channel written to each cycle by the AI Engine at 1 GHz. In order for PL to match the rate of AI Engine, it has to consume twice the data at half the frequency or four times the data at a quarter of the frequency.

AI Engine		PL	
Frequency	**Data per Cycle**	**Frequency**	**Data per Cycle**
1 GHz	32 bit	500 MHz	64 bit
		256 MHz	128 bit

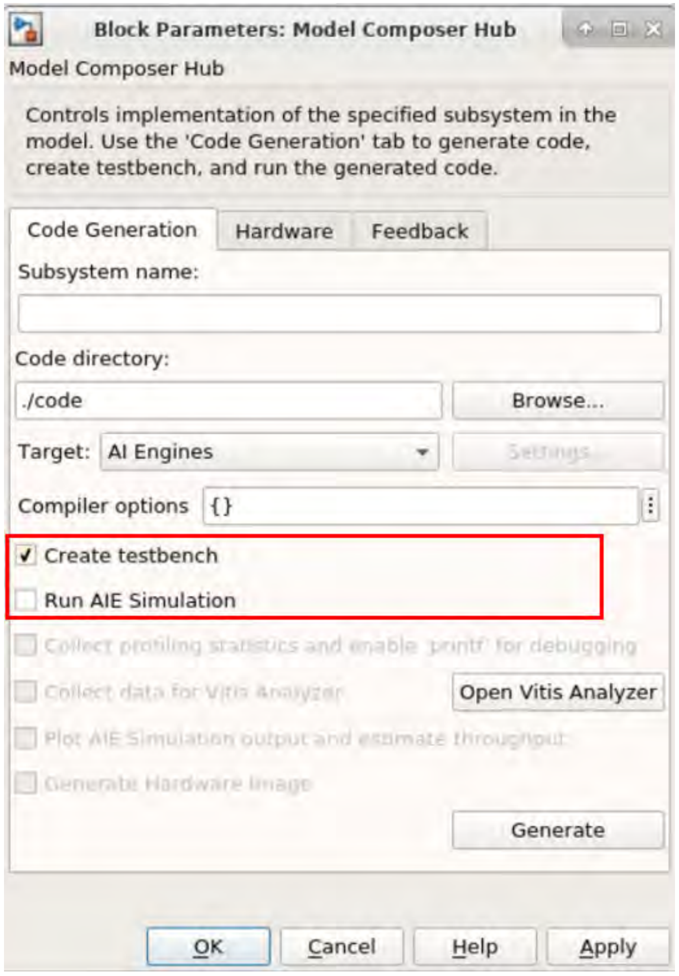
Verification of AI Engine Code

The Versal™ ACAP AIE simulator (`aiesimulator`) models the timing and resources of the AI Engine array accurately while using transaction-level, approximately timed SystemC models for NoC, DDR, PL, and PS. This allows accurate performance analysis of your AI Engine. Model Composer supports verification of the dataflow graph using this AIE simulator.

Model Composer Hub Block for Verification

To verify the AI Engine design, you need to enable the **Create testbench** option on the Model Composer Hub and then choose the simulator options available, as shown in the following figure.

Figure 271: **Model Composer Hub: Testbench and Simulator Options**


Block Parameters: Model Composer Hub

Model Composer Hub

Controls implementation of the specified subsystem in the model. Use the 'Code Generation' tab to generate code, create testbench, and run the generated code.

Code Generation | Hardware | Feedback

Subsystem name:

Code directory: ./code

Target: AI Engines

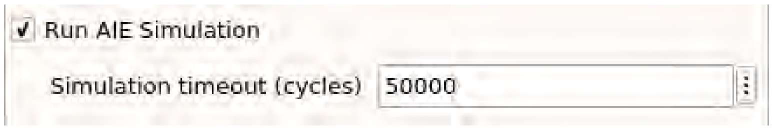
Compiler options {}

- [✓] Create testbench
- [ ] Run AIE Simulation
- [ ] Collect profiling statistics and enable 'printf' for debugging
- [ ] Collect data for Vitis Analyzer
- [ ] Plot AIE Simulation output and estimate throughput
- [ ] Generate Hardware Image

Enabling the **Create testbench** option only logs the test data, or stimulus at the input of your design, and the simulation results as test vectors for later use as "golden" results. By default, no simulator is selected and in order to verify the design, you need to select the **Run AIE simulation** option.

The Simulation timeout value limits the execution to the specified number of cycles. This is necessary because of the finite amount of input data - if the timeout value is not specified, the AI Engine kernels are invoked repeatedly forever (i.e., the graph runs infinitely). To avoid this situation, specify the timeout value for the AIE Simulation as shown in the following figure.

Figure 272: **AIE Simulation: Timeout Value**


☑ Run AIE Simulation

Simulation timeout (cycles) 50000

The default timeout value is set to 50,000 cycles and this value is used to terminate the simulation after the specified number of clock cycles.

When only the **Create testbench** option is enabled, and the code is generated, the structure and contents of the target directory created by Model Composer consists of the following files (not a comprehensive list) in addition to the list mentioned in Code Generation.

Table 24: **Target Directory**

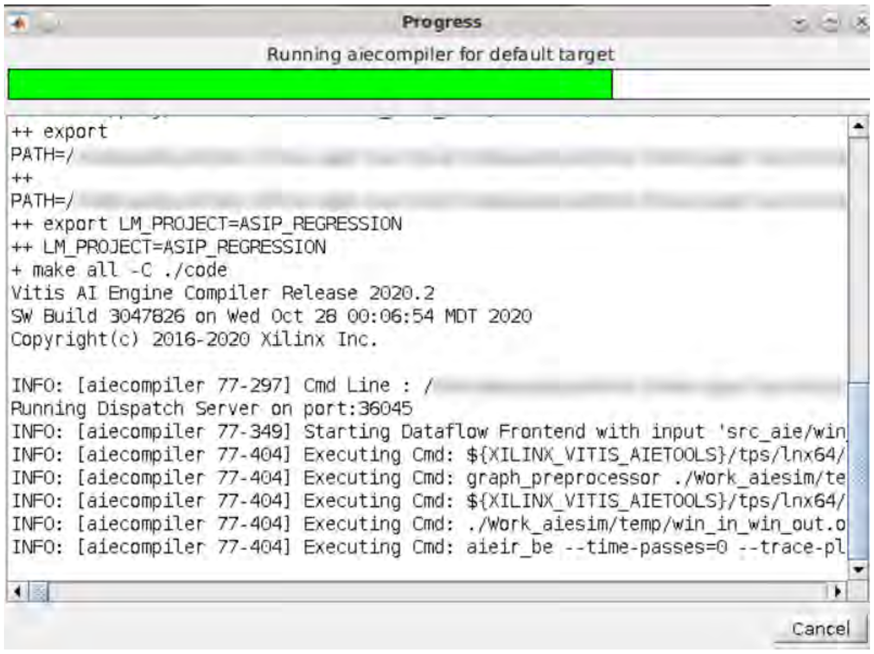
Directory/File	Sub-directory/File	Description
`data/`	`input/`	Contains files that capture the input stimuli from Simulink.
	`reference_output/`	Contains files that capture the simulation output from Simulink.

When the simulator option is enabled, the AI Engine code verification advances in three phases:

1. Compiling the AI Engine graph design.
2. Running simulation using the AI Engine simulator.
3. Verifying the simulation results by comparing the output with the golden reference output.

After clicking **Generate and Run**, you can monitor the compilation, simulation, and verification progress of the AI Engine graph code from the Progress window (see the following figure).

Figure 273: **Graph Code Progress**



```
++ export
PATH=/
++
PATH=/
++ export LM_PROJECT=ASIP_REGRESSION
++ LM_PROJECT=ASIP_REGRESSION
+ make all -C ./code
Vitis AI Engine Compiler Release 2020.2
SW Build 3047826 on Wed Oct 28 00:06:54 MDT 2020
Copyright(c) 2016-2020 Xilinx Inc.

INFO: [aiecompiler 77-297] Cmd Line : /
Running Dispatch Server on port:36045
INFO: [aiecompiler 77-349] Starting Dataflow Frontend with input 'src_aie/win
INFO: [aiecompiler 77-404] Executing Cmd: ${XILINX_VITIS_AIETOOLS}/tps/lnx64/
INFO: [aiecompiler 77-404] Executing Cmd: graph_preprocessor ./Work_aiesim/te
INFO: [aiecompiler 77-404] Executing Cmd: ${XILINX_VITIS_AIETOOLS}/tps/lnx64/
INFO: [aiecompiler 77-404] Executing Cmd: ./Work_aiesim/temp/win_in_win_out.o
INFO: [aiecompiler 77-404] Executing Cmd: aieir_be --time-passes=0 --trace-pl
```

After successful compilation and simulation, Model Composer automatically compares the target output with the golden output and returns the following message in the Progress window (or in the corresponding simulation log files).

```
Comparing simulation results ...
Output data file : data/aiesimulator_output/Out1.txt.mod
reference data file : data/reference_output/Out1.txt
Simulation results MATCH.
**********************************************************
Test PASSED
Verification Complete
```

***Note*:** In some scenarios, the simulator output produces fewer samples when compared with the golden output or vice-versa. In such cases, the test result will be still show as 'PASS.' This indicates that the first 'n' lines from the simulation output matches the first 'n' lines of the golden output and the results are partially matched. The `.diff` file in the corresponding simulator output captures any difference with the reference output.

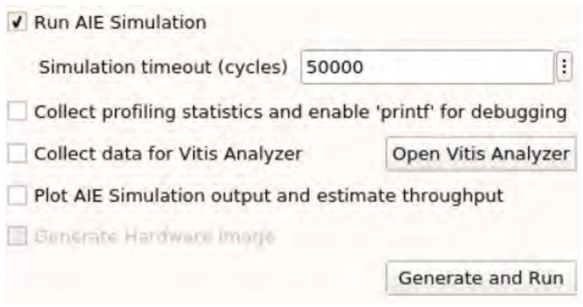
Table 25: **Target Directories**

File Name/ Parent Directory	Filename/Sub-directory	Description
`data/`	`aiesimulator_output/`	Contains `.txt` files that captures the output of the AIE simulation. In addition to the actual output file, the `.diff` file is also generated to capture any difference from the golden result.

In addition to this, `aiecompiler` writes various configuration and binary files to the `Work_aiesim` directory. For more information on the structure and contents of the directory specific to the compilation, refer to the *Versal ACAP AI Engine Programming Environment User Guide* (UG1076).

When the **Run AIE Simulation** option is enabled, the features shown in the following figure are available. You can select one or all options, then click **Generate and Run**.

Figure 274: **Run AIE Simulation**


[✓] Run AIE Simulation

Simulation timeout (cycles) 50000

[ ] Collect profiling statistics and enable 'printf' for debugging

[ ] Collect data for Vitis Analyzer

[ ] Plot AIE Simulation output and estimate throughput

[ ] Generate Hardware Image

Profiling Statistics and Event Tracing

You can obtain profiling data when you run your AIE simulation. Analyzing this data helps you gauge the efficiency of the kernels, the stall and active times associated with each AI Engine, and pinpoint AI Engine kernels whose performance may not be optimal. This also allows you to collect data on design latency, throughput, and bandwidth. In addition to this, you can do event trace using a formatted `printf` statement in the code for printing debug messages. To acheive this, you should enable the option **Collect profiling statistics and enable 'printf' for debugging** in the Model Composer Hub block.

IMPORTANT! *Using this option generates a `run_summary` file which is written to the `aiesimulator_output` folder.*

IMPORTANT! *Enabling this option may slightly increase the overall AIE simulation time.*

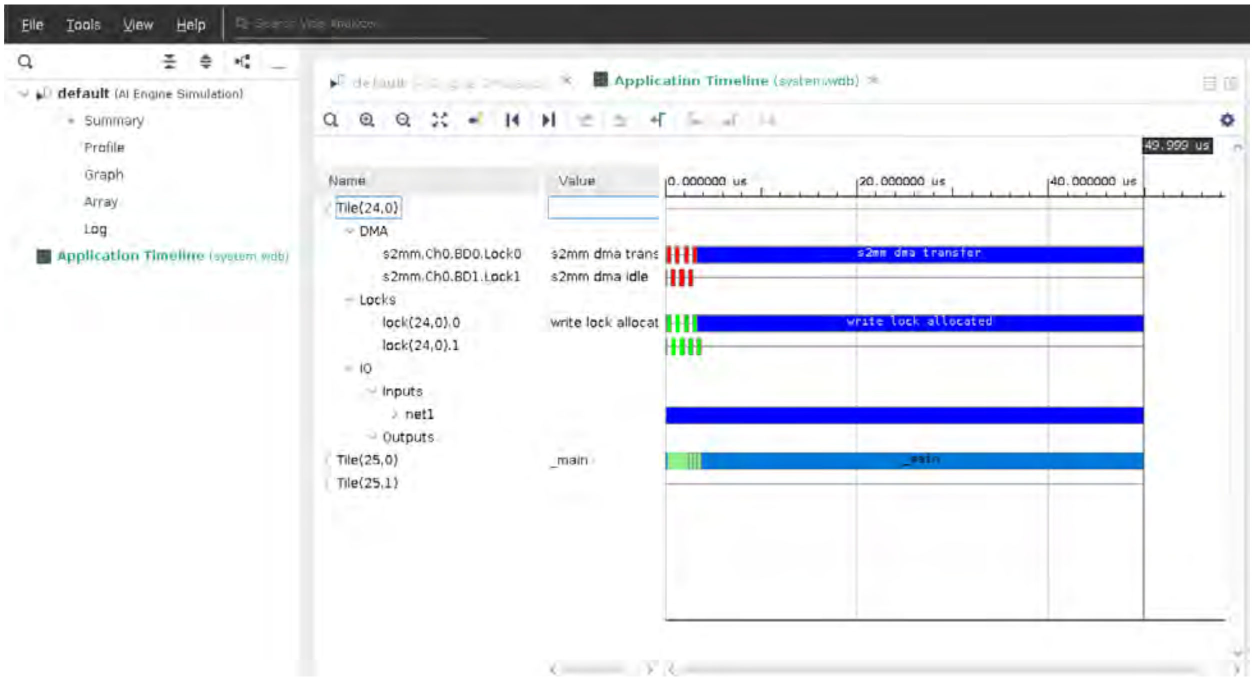
Viewing Results in the Vitis Analyzer

The Vitis™ software platform analyzer is a utility that allows you to view and analyze the reports generated while building and running the application. It is intended to let you review reports generated by both the Vitis compiler when the application is built, and the Xilinx Runtime (XRT) library when the application is run. The Vitis analyzer can be used to view reports from Vitis integrated design environment (IDE).

During the simulation of the AI Engine graph, the `aiesimulator` writes a summary of the simulation results called `default.aierun_summary`. This can be viewed in the Vitis analyzer. The summary contains a collection of reports and diagrams reflecting the state of the AI Engine application.

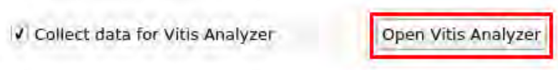
Model Composer integrates the Vitis analyzer utility. This can be invoked by enabling the **Collect Data for Vitis Analyzer** option from within the Model Composer Hub block, along with the AIE Simulation. When the simulation completes, Model Composer automatically reads the `system.wdb` file which is generated during the `aiesimulator` run and invokes the Application Time line Window in Vitis Analyzer as shown.

Figure 275: **Application Timeline**



From the report navigator you can view other available reports, such as Summary, Profile, Graph, Array, and Log.

***Note*:** You can relaunch the Vitis Analyzer by clicking **Open Vitis Analyzer** as shown in the following figure. This option can be used to invoke the Vitis Analyzer tool, only when the AIE Simulation has been ran at least once after enabling the **Collect data for Vitis analyzer** option.



In Model Composer, you can launch Vitis analyzer from the MATLAB command window using the following command.

```
xmcOpenVitisAnalyzer('file')
```

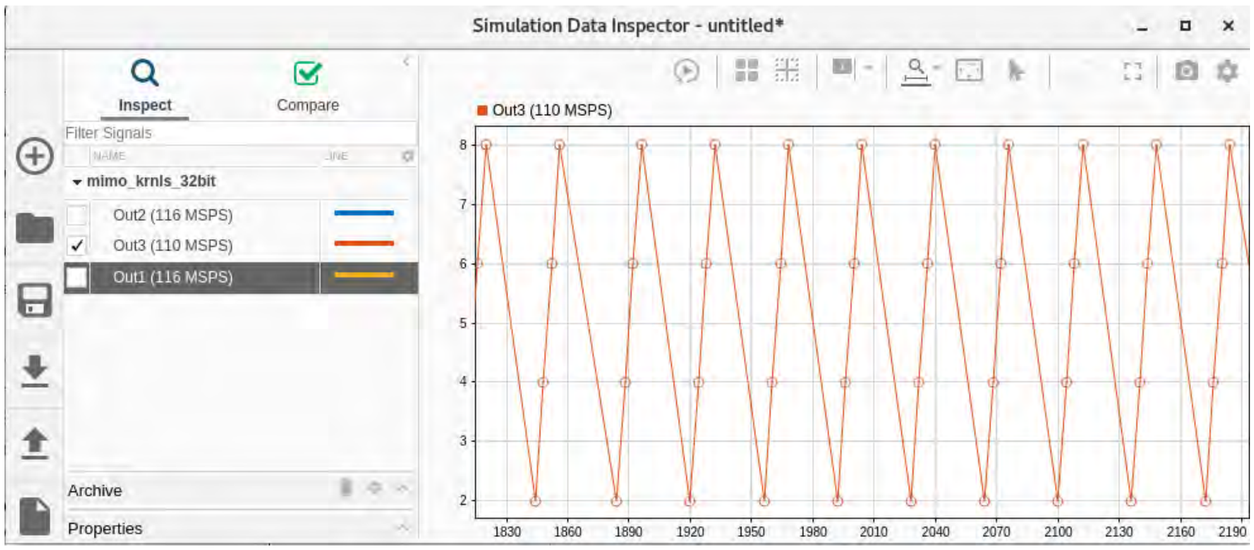
When this command is given without a `<file>` option, Model Composer looks up all `default.aierun_summary` and `system.wdb` files in the current target directory and invokes the Vitis analyzer to display the results in the summary page.

For more information on the Vitis analyzer, refer to the Using Vitis Analyzer topic in *Vitis Unified Software Platform Documentation* (UG1416).

Plotting AIE Simultion Output Data and Calculating Throughput

Model Composer provides the capablity to log the simulation data and visualize the output of an AI Engine subsystem by integrating the Simulink 'Simulation Data Inspector' feature. It also calculates the throughput for each output port of the AI Engine subsystem. To achieve this, enable the option **Plot AIE Simulation output and estimate the throughput** from the Model Composer Hub block. When the AIE simulation completes, the tool automatically brings up the Simulation Data Inspector window reflecting the outputs of the AI Engine subsystem as shown in the following figure.

Figure 276: **Simulation Data Inspector**



The Simulation Data Inspector displays available data in the Inspect pane. To plot a signal, select the check box next to the signal. You can modify the layout and add different visualizations to analyze the simulation data. You can also get the throughput information for each port from the Inspect pane. If required, you can re-open the Simulation Data Inspector from the Simulink® Editor toolbar using the **Simulation Data Inspector** button.

For more information on using the Simulink Simulation Data Inspector, visit https://in.mathworks.com/help/simulink/ug/populate-sdi-with-your-data.html.

IMPORTANT! *Model Composer can only plot the outputs of the AI Engine subsystem automatically.*

Running Simulation using the Makefile

You may want to edit the graph code, for example, to add or modify constraints, or explore the graph code by editing the kernel configuration parameters (like window size or window margin etc.), and re-compile. Model Composer generates a Makefile allowing you to easily accomplish this by running the following command.

```
make all
```

This command compiles an AI Engine graph using the default `aiecompiler` target and runs the simulation using `aiesimulator`. It also compares the output of the simulator with the golden output. This command also launches the Vitis analyzer, based on the option selected in the Model Composer Hub block.

To do this, you need to source the `settings64.csh` or `settings64.sh` from `<MODEL_COMPOSER_INSTALLATION_DIRECTORY>/Model_Composer/<VERSION>`

When the code is generated without the **Create testbench** option checked, the `aiesimulator` settings in Makefile are as follows.

Default aie simulator settings

```
############################################################################
####
# aiesimulator settings
############################################################################
####
AIE_SIM := aiesimulator
AIE_SIM_TIMEOUT := 50000
AIE_OUTPUT_DIR := $(DATA_DIR)/aiesimulator_output
AIESIM_FLAGS := --profile
LAUNCH_VITIS_ANALYZER :=
```

Observe the Timeout value is `50000` which is default and `LAUNCH_VITIS_ANALYZER` is not set to `TRUE`. To run simulation for different timeout values with the Vitis analyzer enabled, you may need to manually edit the Makefile as follows.

Modified Settings

```
AIE_SIM_TIMEOUT := <Value>
LAUNCH_VITIS_ANALYZER := true
```

When the code is generated with the **Create testbench** option selected, and and the simulator option is enabled, then the modifications done in the Model Composer Hub block for **Simulation Timeout** and **Collect Data for Vitis Analyzer** are automatically reflected in the Makefile.

Hardware Validation Flow for AI Engines

Introduction

The hardware validation flow for AI Engines in Vitis Model Composer provides a methodology to verify AI Engine-based applications on Xilinx hardware (Versal devices). Vitis Model Composer provides a set of blocks in the Simulink library browser that make it easy to develop applications for Xilinx devices by integrating HDL/HLS blocks for the Programmable Logic (PL) and AI Engine blocks for the AI Engine array. Vitis Model Composer can be used to create complex systems targeting the PL and AI Engine array at the same time. The complete system can be simulated in Simulink followed by code generation (HDL for the PL and C++ graph for the AI Engine array). Vitis Model Composer also provides the option to generate a hardware image (BOOT.BIN) targeting a specific platform for the Simulink model. This hardware image can then be run on a board to verify whether the results from hardware match with the simulation outputs.

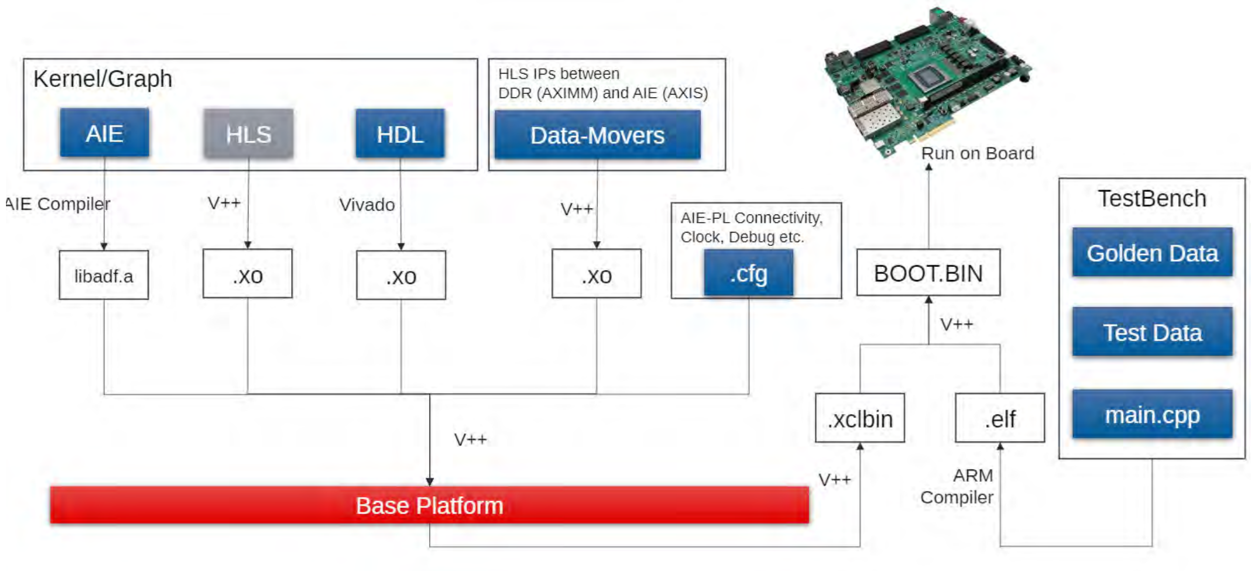
IMPORTANT! *Vitis Model Composer supports integration of blocks from the AI Engine library with the HDL library to generate a hardware image. However, to integrate with HLS C/C++ code, you should import using HLS Kernel block, and not the blocks from HLS library.*

High-Level Flow for Generating a Hardware Image

Vitis Model Composer generates the code necessary to create the hardware image (BOOT.BIN) as part of code generation process. This includes various scripts, PL DataMover IPs, a host application that runs on the PS, input test data, and golden output data. The `run_hw.sh` sets up the necessary environment and then runs a Makefile (generated within the `<code-generation-directory>/run_hw` directory) to generate the BOOT.BIN.

For more information on the various files generated during the hardware validation flow, refer to Output Files. The following figure shows the high-level flow of the BOOT.BIN generation.

Figure 277: **BOOT.BIN Generation Flow**



The following table describes the high-level steps executed via Makefile, that is generated within the `<code-generation-directory>/run_hw` directory.

Table 26: **Makefile Steps**

Makefile Step	Description
prepdir	This step copies the necessary files from the `<Target_directory>` to inside the `run_hw/` directory. It also prepares input test data and golden output data such that it can be passed on to the hardware and exchanged via the PL DataMover IPs.
datamover_kernels	This step compiles PL DataMover kernels (auto-generated `mm2s.cpp` and `s2mm.cpp`) to generate corresponding `XO` files using v++.
hls_kernels	If the design contains HLS kernels, this step compiles these to generate corresponding `XO` files. The script `HlsKernelXOGen.sh` drives the generation of XOs using v++. If you have specified any compilation flag for the HLS kernel, then these are provided to the v++ compiler during the corresponding kernel compilation
hdl_kernels	If the design contains HDL kernel, this step compiles these to generate corresponding XO file. The script `HDLKernelXOGen.tcl` drives the generation of XO using Vivado.
graph	This step compiles the generated AIE graph to produce `libadf.a` using AIE Compiler. In iterative workflows (using incremental modifications to the AIE code and regenerating the hardware image), the graph may be recompiled to produce a new `libadf.a`. However, the actual AIE shim solution may remain the same as in the previous iteration. The Makefile detects any changes in the generated AIE shim solution and updates the timestamp on a file named `aiesoldiff`, which is later used to detect whether the platform needs to be regenerated.
platform	This step builds the platform by linking kernels and graphs to the base platform to produce an XSA and XCLBIN. This is also where the generated `<AIE_Subsystem>.cfg` file is used, which contains the connectivity between AIE IP and rest of the platform.

Table 26: **Makefile Steps** *(cont'd)*

Makefile Step	Description
application	This step compiles the host application to generate `main.elf`. The generated script `genPSMain.sh` writes a header file `xmc_ps_main.h` which is included in the host application `main.cpp`. This generated file `xmc_ps_main.h` contains the driver details and base address information for the PL Datamover kernels, and the input test data and output golden data sample sizes, as produced during simulation.
pdi	This step runs the v++ packager to generate the hardware image (BOOT.BIN). The generated BOOT.BIN can be run on hardware to verify the results with the golden output data from simulation. For more information on step on running BOOT.BIN on hardware, refer to Running BOOT.BIN on Hardware. If the Makefile is run with the TARGET set as `hw_emu`, it runs the v++ packager to generate a script `launch_hw_emu.sh` to run hardware emulation on the design.
run_emu	When the Makefile is run with the TARGET set as `hw_emu`, this step runs the hardware emulation on the design by executing the generated script `launch_hw_emu.sh`. If the Makefile is run with the TARGET set as `hw` (default target), then this step is not applicable.

Generating Hardware Image

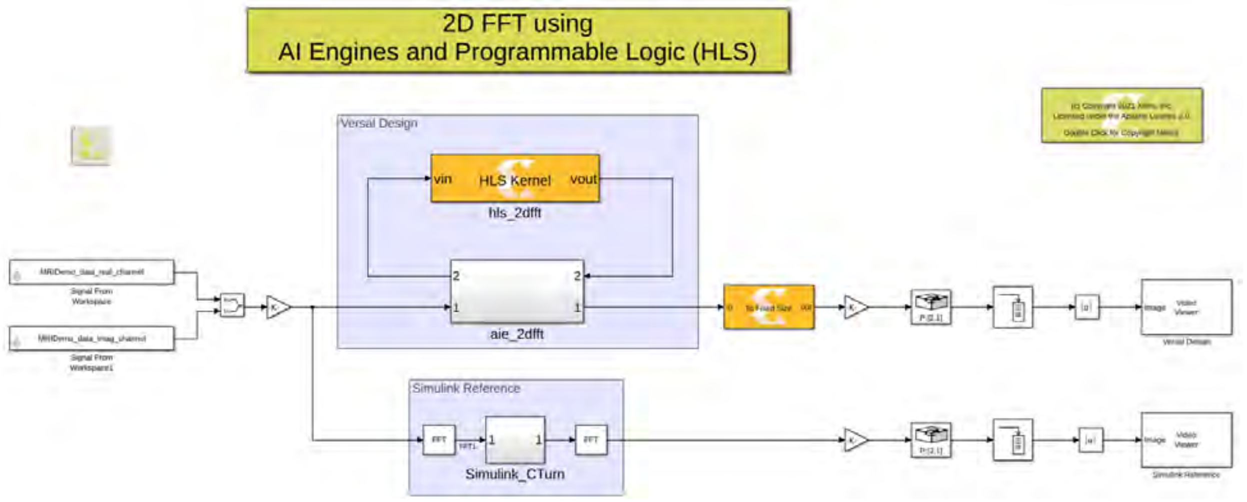
Vitis Model Composer automates the flow of generating the hardware image out of a design containing AI Engine, HDL, and HLS kernel blocks. These steps differ slightly based on the communication among the AI Engine and different Programmable Logic (HLS kernel or HDL) blocks. This section provides details for generating a hardware image for:

- A design with AI Engine and HLS kernel blocks.
- A design with AI Engine and HDL blocks.

Design with AI Engine and HLS Kernel Blocks

This section discusses generating the hardware image for a design that contains AI Engine and HLS kernel blocks. You can connect the HLS kernel block alongside the AI Engine blocks and still generate the hardware image.

Figure 278: **2D FFT using AI Engines and PL (HLS)**



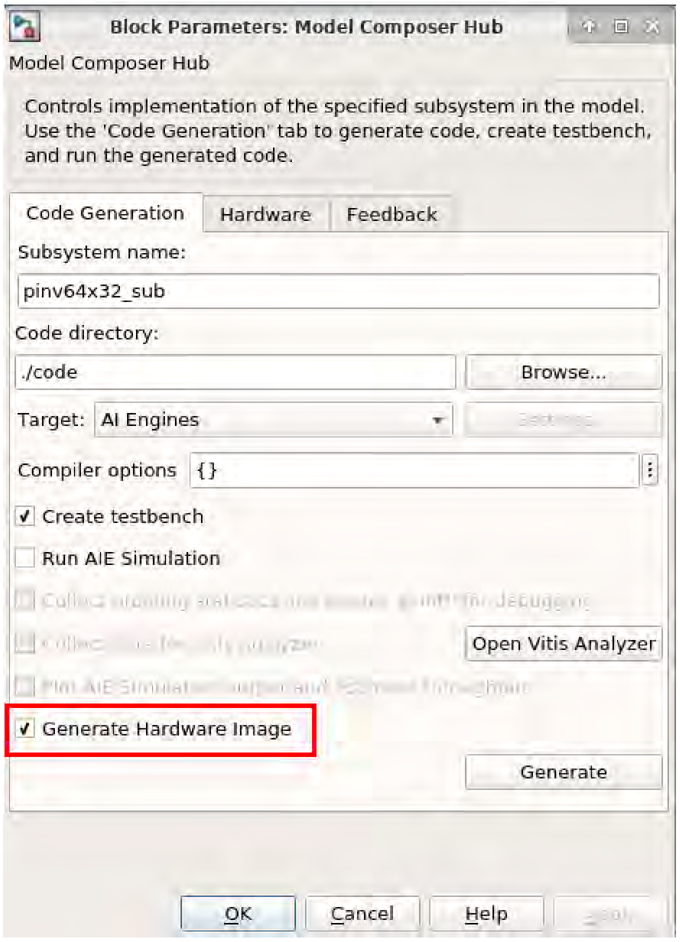
Consider the design shown in the previous figure, where the HLS kernel block is connected between two AIE kernel blocks. This example reads in complex data from the MRI signal, performs a 2D FFT algorithm on this data using AIE and HLS blocks and displays the corresponding image of the brain at output. As depicted in the figure, the AI Engine kernels are joined to form an AI Engine subsystem which will be targeted to the AI Engine portion of the Versal hardware, and the HLS kernel block is targeted to the PL portion. This example is available in GitHub.

To generate a hardware image and validate the design on hardware, Vitis Model Composer provides an option in the Model Composer Hub block. You need to enable the **Generate Hardware Image** check box from the Model Composer hub block using the following steps:

1. On the Code Generation tab within the Hub block, specify the Target as **AI Engines**.
2. Select the **Create testbench** check box.
3. On the Hardware tab, select **Specify Platform** from the drop down menu and specify the full path of the valid platform.

   Note**:** Code for the hardware validation flow will not be generated unless a valid platform has been specified in the Hardware tab of the Model Composer Hub block.
4. Go back to the Code Generation tab and select the **Generate Hardware Image** check box.
5. Click **Apply** and then **Generate**.

Figure 279: **Generate Hardware Image**


Block Parameters: Model Composer Hub

Model Composer Hub

Controls implementation of the specified subsystem in the model. Use the 'Code Generation' tab to generate code, create testbench, and run the generated code.

Code Generation | Hardware | Feedback

Subsystem name: pinv64x32_sub

Code directory: ./code

Target: AI Engines

Compiler options {}

☑ Create testbench

☐ Run AIE Simulation

☑ Generate Hardware Image

***Note*:** You can optionally select the **Run AIE Simulation** check box. This is not necessary for Hardware Validation flow, but selecting this option allows you to verify the simulation results of your design using AIE Simulator.

At the end of successful code generation, with **Generate Hardware Image** selected in the hub block, BOOT.BIN will be generated. You will see the following in the Simulink progress window.

```
INFO]: Bootimage geneated successfully
...
...
************************************************
XMC_RUNHW_INFO: Finished running run_hw.sh
************************************************
SUCCESS
```

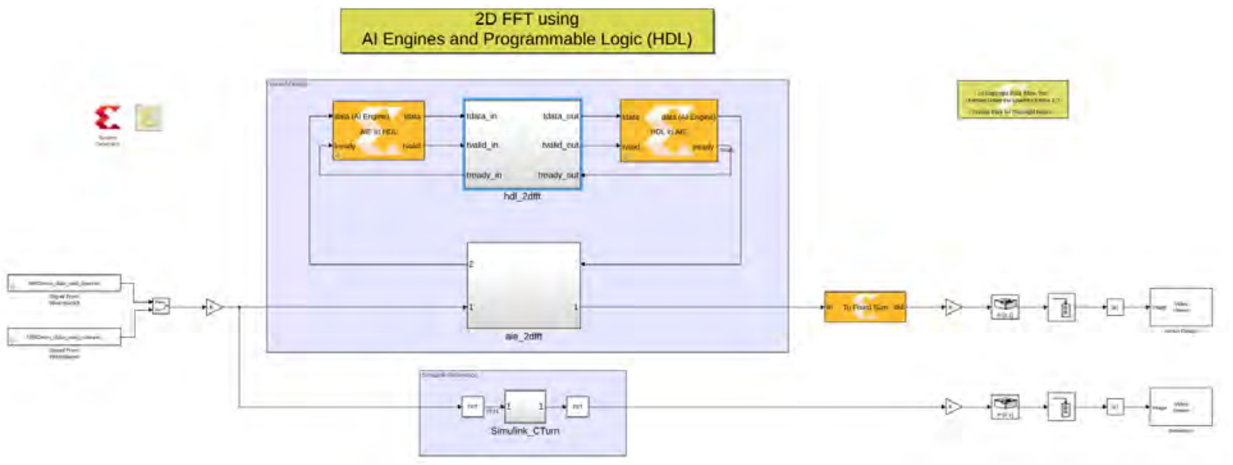
This boot image can then be run on hardware to validate simulation outputs with the results from hardware. For more information on running BOOT.BIN on hardware see Running BOOT.BIN on Hardware.

Design with AI Engine and HDL Blocks

This section discusses generating the hardware image for a design that contains AI Engine and HDL blocks. You can connect the blocks from HDL Library alongside the AI Engine blocks and still generate the hardware image as described in the Design with AI Engine and HLS Kernel Blocks. However, additionally you need to perform an extra step to generate the code for HDL blocks prior to generating the hardware image using Model Composer Hub block.

To understand more, consider the same 2D-FFT example, but now implemented using an HDL block as shown in the following figure.

Figure 280: **2D FFT using AI Engines and PL (HDL)**



Here, the subsystem `aie_2dfft` is the AI Engine portion of the design and the `hdl_2dfft` subsystem is the Programmable Logic (HDL) portion of the design.

This example is available in GitHub.

As depicted in the previous figure, the design uses the System Generator Token block to generate the code for the HDL portion of the design. The following sequence should be followed for hardware image generation:

1. Generate code for HDL blocks from System Generator token.
2. Generate code for AI Engine blocks from the Model Composer Hub block.

The hardware image generation from the Model Composer Hub block depends on the generated HDL netlist. This necessitates the above sequence to be followed.

Generating HDL Code

Open the System Generator block and specify the following options:

- For Part select **Versal AI Core Series→xcvc1902 → -2MP-e-S → vsva2197**.

- Select **IP Catalog** from the list of Compilation targets.
- Click **Settings** next to the IP Catalog selection box to launch IP Catalog Settings dialog and specify `Xilinx.com` in the Vendor edit box. Click **OK**.
- Select **Verilog** from the drop-down menu under Hardware description.
- Under Target directory, specify **./netlist**.

  Note: Target directory must be **./netlist** for Hardware Validation flow. This limitation will be relaxed in future releases.

The code generation for the AI Engine subsystem and the hardware validation flow remains the same as explained in Design with AI Engine and HLS Kernel Blocks.

Generating the Hardware Image outside the MATLAB Environment

When **Generate Hardware Image** is selected from the Model Composer Hub block and the **Generate and Run** button is clicked, the process can take some time, especially when the BOOT.BIN is being built for the first time. This may stall that particular MATLAB session for any other work. In this case it might be desirable to generate the BOOT.BIN outside of MATLAB, such that the MATLAB session remains free for other work.

To generate BOOT.BIN outside of the MATLAB session, follow these steps:

1. Generate code without selecting the **Generate Hardware Image** check box in the Hub block. This means:
   - ◦ If the design contains HDL blocks, generate the netlist from the System Generator token block.

     ◦ Generate code for AIE and HLS blocks from the Model Composer Hub blocks. Specify a valid platform from the Hardware tab and select the Target as **AI Engines**. Select the **Create testbench** check box on the Code Generation tab, but do *not* select the **Generate Hardware Image** check box.
2. Open a linux terminal and ensure the `$XILINX_VITIS` environment variable is set.

   Note: Source the script `settings64.sh` from `<Model Composer Install directory>/Model_Composer/<Version>/settings64.sh` in order to setup the XILINX_VITIS environment variable. (For `cshell`, source the `settings64.csh` file)
3. cd to `<code-generation-directory>/run_hw` directory.
4. From the linux terminal run `./run_hw.sh`.

Improving Platform Build-Time performance

When generating the hardware image, the step that consumes most time is the Platform step in the Makefile, which builds the hardware platform by linking kernels and AIE graph to a base hardware platform. If you have access to a machine with more processors, you can parallelize the jobs by running them on multiple processors. To do that, update the `NPROC` value (the default is 4) in `run_hw.sh` and (re)run. Use caution while working on a shared machine as all the processors may not be available for use.

Developing AI Engine Code Incrementally and Validating the Hardware Flow

In iterative workflows (using incremental modifications to the AIE code and regenerating the hardware image), the graph may be recompiled to produce a new `libadf.a`. However, the actual AIE shim solution may remain the same as in the previous iteration. The Makefile detects any changes in the generated AIE shim solution and updates the timestamp on a file named `aiesoldiff`, which is later used to detect whether the platform needs to be regenerated. If the AIE shim solution is not different, then the platform will not be rebuilt, assuming the other dependencies of the platform target remain unchanged.

For example, when you try to change the datatype of the AI Engine kernel interface or functionality of the kernel, only the graph gets recompiled to produce a new `libadf.a`. However, changing the number of ports in the kernel interface or modifying the PLIO width of the ports leads to the change in the AIE SHIM solution which, in turn, rebuilds the platform.

Output Files

The following table provides descriptions of the directories and files that are generated by the AI Engine code generation flow in Model Composer when hardware flow is enabled. The location of the files and directories shown in this table are relative to the `Target` directory specified in Model Composer Hub block for code generation.

Note: This list only describes files/directories that are specific to the hardware validation flow. For information on various sub-directories in the `<Target directory>/` refer to Output Directory.

Table 27: **File/Directory Descriptions**

File/Directory Name	File	Description
`src_pl_datamover/`	`mm2s.cpp`	This is a PL DataMover IP, automatically inferred for input data to the AIE subsystem. This converts AXI Memory-Mapped data from external DDR Memory into AXI Stream for the AI Engine Array.
	`s2mm.cpp`	This is a PL DataMover IP, automatically inferred for output data from the AI Engine subsystem. This converts AXI-Stream data from the AI Engine Array into AXI Memory-Mapped data

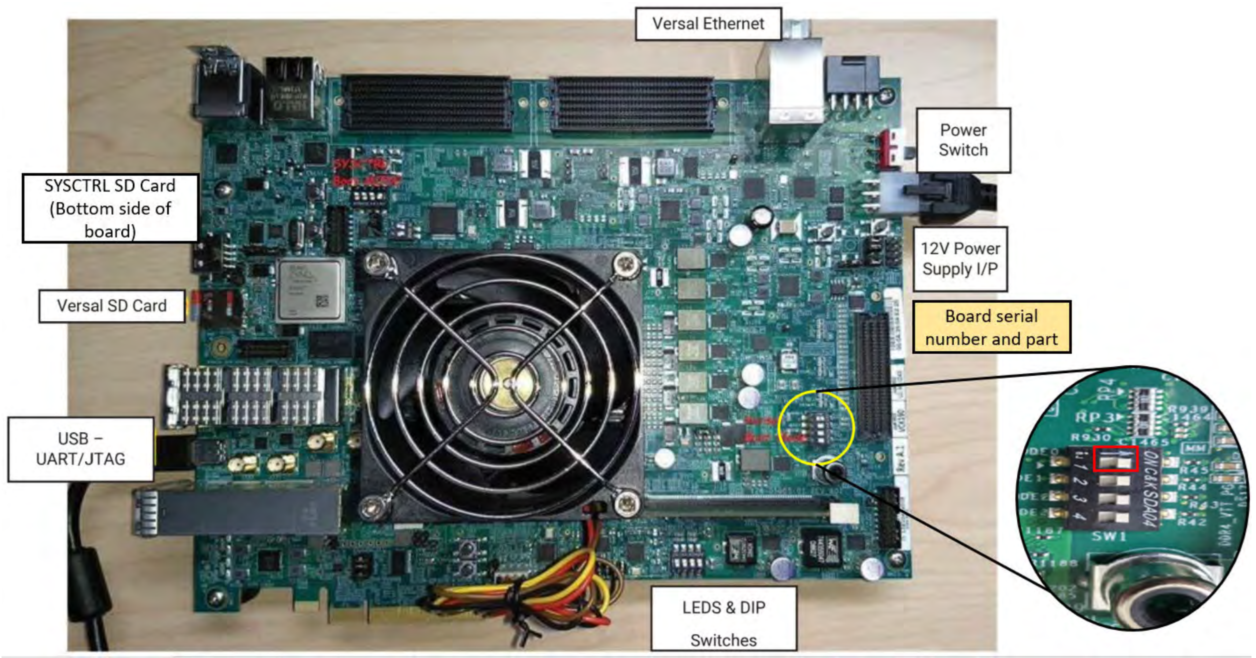
Table 27: **File/Directory Descriptions** *(cont'd)*

File/Directory Name	File	Description
src_ps/	`main.cpp`	This is the host application that runs on the PS. It instantiates and runs the AI Engine dataflow graph and supplies the simulation test data for comparison with results after running on hardware.
	`genPSMain.sh`	This script generates the `xmc_ps_main.h` header file which is included from host application (i.e., `main.cpp`). `xmc_ps_main.h` lists the following: • Addressing information for PL Datamover IPs (MM2S and S2MM) interfaces, as obtained from the generated hardware drivers. • Input and Output sample sizes, as obtained from the input test data and output golden data files while running simulation in Model Composer.
`hdl_xo/`	`HdlKernelXOGen.tcl`	This is the TCL file that gets sourced in Vivado. It is used to generate the XO for the HDL kernel present in the design. This file is generated only if there are HDL blocks present in the design.
`run_hw/`	`Makefile`	This is the Makefile specifying the steps to run the design on hardware. TARGET can be `hw_emu` or `hw`.
	`run_hw.sh`	Top-level script to run the Makefile. Default TARGET is `hw`.
	`prepData.sh`	Preprocessing script which prepares generated data from Model Composer for the `run_hw` directory.
	`prepData.py`	Preprocessing script that gets invoked from within `prepData.sh`. Translates input test data and output golden data files into a suitable format for hardware, based on datatypes and converts them into suitable header files which are included in the host application.
	`HlsKernelXOGen.sh`	Script to compile HLS kernels (if any) present in the model to generate XOs.
	`graph.mk`	Included from the Makefile; lists AI Engine sources.
	`hls.mk`	Included from the Makefile; lists HLS sources.

Running BOOT.BIN on Hardware

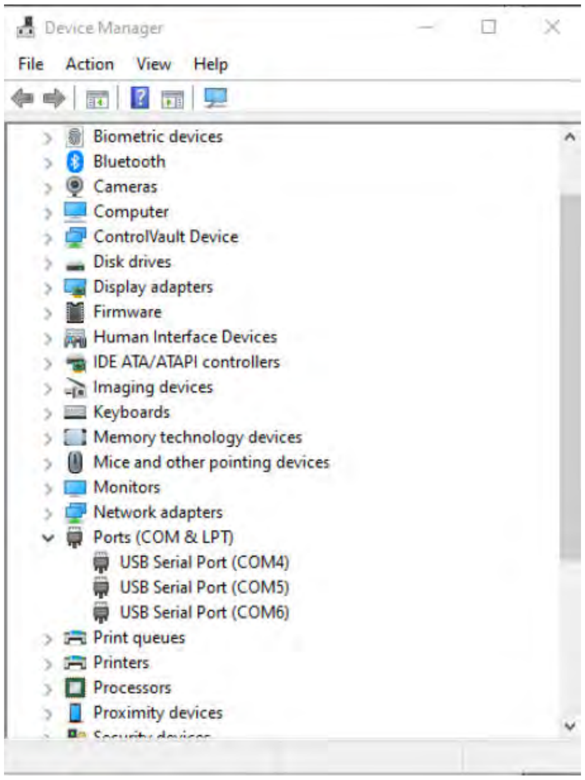
In this section you will learn how to manage the board settings, make cable connections, connect the board through your system, and program the BOOT.BIN on a Versal device.

1. Connect the power cable to the board.
2. Connect a USB Micro cable between the host machine and USB JTAG connector on the target board. This cable is used for USB to serial transfer.
3. Ensure that the `SW1` switch is set to the JTAG boot mode as shown in the following figure, then power on the VCK190 board.



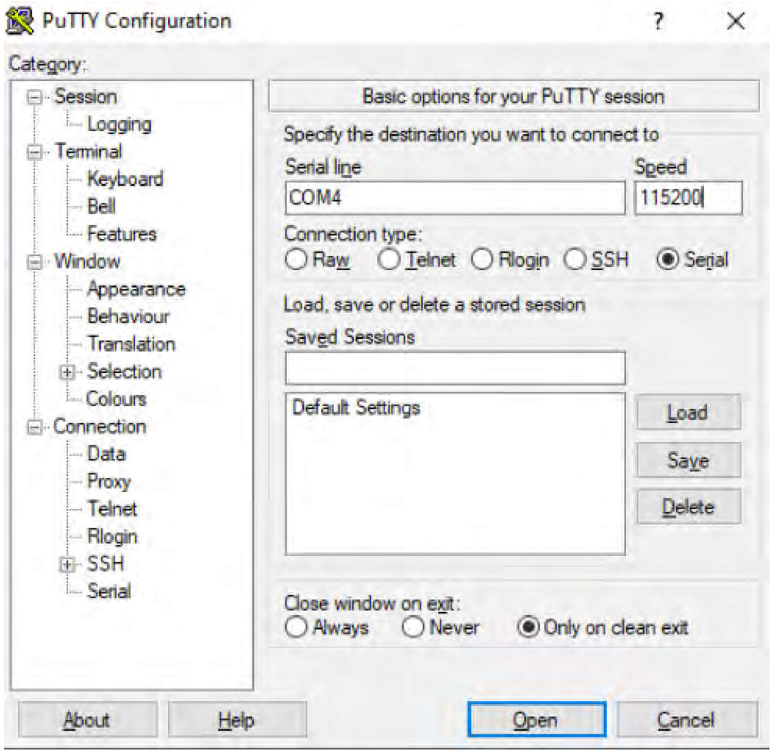
Once your board is set up, program the device as follows.

4. In Windows 10, click the search box on the taskbar and type `Device Manager` , then select from the Menu.
5. When the board is powered ON and connected to your machine through the USB interface, determine the COM ports that are between the VCK190 board and your computer from Windows Device Manager.



Biometric devices
Bluetooth
Cameras
Computer
ControlVault Device
Disk drives
Display adapters
Firmware
Human Interface Devices
IDE ATA/ATAPI controllers
Imaging devices
Keyboards
Memory technology devices
Mice and other pointing devices
Monitors
Network adapters
Ports (COM & LPT)
USB Serial Port (COM4)
USB Serial Port (COM5)
USB Serial Port (COM6)
Print queues
Printers
Processors
Proximity devices

6. Use a Terminal application (Teraterm or Putty) to open up COM port interfaces on these ports (COM4, COM5 and COM6 in the above case at 115200 baud rate).



Specify the destination you want to connect to
Serial line: COM4
Speed: 115200
Connection type: Raw Telnet Rlogin SSH [✓ Serial]
Load, save or delete a stored session
Saved Sessions
Default Settings
Close window on exit: Always Never [✓ Only on clean exit]

***Note*:** Ensure the Connection type is set to **Serial**.

7. Set up board with XSDB:

    a. Launch XSDB using the `xsdb.batch` file from the Vitis installation directory as shown.

    ```
    <Xilinx Install Directory>\Vitis\<Version>\bin\xsdb.bat
    or
    <Xilinx Install Directory>\Vivado\<Version>\bin\xsdb.bat
    ```

    b. From the XSDB prompt, run the following commands:

    ```
    connect
    ta
    ta 1
    ```

    c. From within the XSDB prompt, navigate to the directory where hardware device image has been generated. In general, it is in `<code-generation-directory>/run_hw/src_ps/BOOT.BIN`.

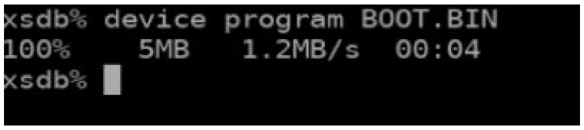
    ```
    cd <code-generation-directory>/run_hw/src_ps/BOOT.BIN
    ```

8. Program the device and run:

    a. From the XSDB prompt, run the following command:

    ```
    device program BOOT.BIN
    ```

    You will see following message in the XSDB prompt if the device program is successful.



```
xsdb% device program BOOT.BIN
100%     5MB   1.2MB/s  00:04
xsdb%
```

    b. This will run the design on the board and you should see the log similar to the following. Here, the hardware results are compared with the golden results.

    ```
    ******************************** Test Results
    ********************************

    ****** Model Composer and Hardware outputs match for all 40944
    samples for output signal Out1 *******

    ****** Test passed ********
    ```

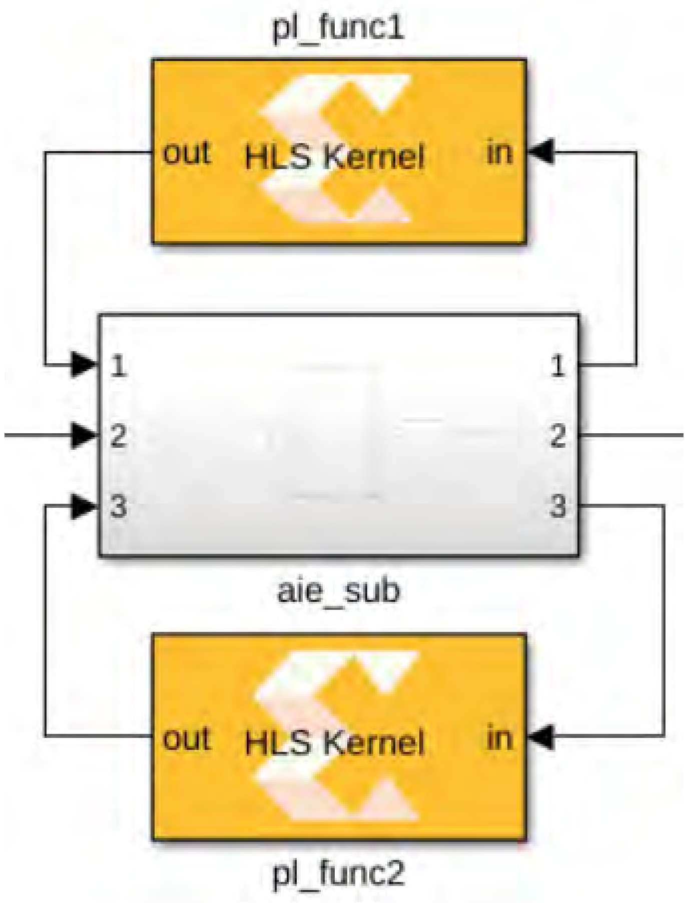
    c. To re-run the results, run following commands:

    ```
    rst -system
    device program BOOT.BIN
    ```

Design Considerations

- Hardware validation flow for designs with only HLS or HDL blocks is not currently supported.
- For designs with HDL blocks, the 'HDL Netlist' generation directory must be `./netlist`, and the sequence of code generation must be followed as explained in the section Design with AI Engine and HDL Blocks.

- Designs with HLS kernel blocks connected to other HLS kernel blocks are not currently supported. Designs with HLS kernel blocks connected to AIE DUT are supported, however you can connect multiple HLS kernels to the AIE DUT. For example, consider the topology as shown in the following figure.



  Here, two HLS kernels are connected to the different ports of the `aie_sub`. This is supported for the hardware validation flow.

- You can only use HLS kernel blocks to import C/C++ code (for PL) and to connect with AI Engines. The blocks from the HLS library are not allowed to connect and co-simulate.

Connecting AI Engine and Non-AI Engine Blocks

AI Engine/Programmable Logic Integration

An AI Engine kernel written using specialized intrinsic and imported into Model Composer can be used as part of a larger Versal™ ACAP system design. In addition to kernels operating on the AI Engines, you can specify kernels to run on the programmable logic (PL) region of the device. The PL kernels can be written using RTL or HLS C/C++ functions. The connection between AI Engine and the PL block is routed through a physical channel interface tile and conceptually the data width of the connections are 32 bits, 64 bits or 128 bits.

Model Composer allows connecting an AI Engine kernel to a HLS PL kernel only if the data types and complexities of these port matches. If the datatypes or complexities of the port of the AI Engine kernel and the port of the PL kernel do not match, an interface blocks should be used to reconcile the discrepancy.

This chapter discusses interconnecting HDL blocks or HLS C/C++ functions with AI Engine kernels:

- Interconnecting AI Engine and HDL Blocks
- Interconnecting AI Engines and HLS Kernels

Interconnecting AI Engine and HDL Blocks

The HDL blockset in the Xilinx® toolbox contains the common DSP building blocks such as adders, multipliers, and registers. It also includes a set of complex DSP building blocks such as FFTs, filters and memories. Model Composer automatically compiles designs into low-level representations (i.e., RTL) which can be targeted to programmable logic. The Versal hardware allows for connecting AI Engine kernels and PL kernels. However, HDL and AI Engine domains are incompatible in at least two aspects:

- The HDL domain is cycle accurate, whereas the AI Engine domain is only bit accurate. A Model Composer HDL design may cause back pressure or may not have a valid output and these are managed through designs with `Tvalid` and `Tready` signals of AXI Stream ports.

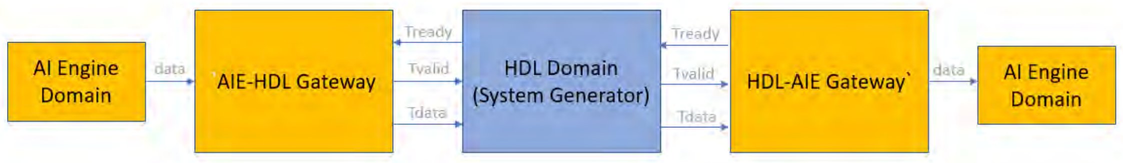
- Model Composer accepts only scalar inputs, whereas AI Engine blocks work with variable sized vector signals.

In order to properly manage the sampling times across two domains and simulate the heterogeneous system with both PL (modeled with HDL blocks) and AI Engine, Model Composer provides interface blocks in the AI Engine library to connect from AI Engine to HDL blocks and vice versa.

- AIE to HDL - This block connects AI Engine to HDL blocks using an AXI4-Stream-like interface.
- HDL to AIE - This block connects HDL to AXI4-Stream blocks using AXI4-Stream-like interface.

You can find these blocks in `Xilinx Toolbox/AI Engines/Interfaces` library.

Figure 281: **Interface Blocks**



As discussed, AIE-HDL and HDL-AIE blocks have `Tvalid` and `Tready` ports as depicted in the previous figure. The gateway from the AI Engine to the HDL domain performs the unbuffer operation, meaning that the HDL domain will run at a different rate than the AI Engine domain. For example, if the AI Engine domain is producing 16 samples at every clock, the HDL domain must run at least 16 times faster to process the data. However, in some cases, the HDL domain needs to run even faster because the HDL domain, being a cycle accurate domain, may not consume one sample per clock. For example, if the HDL domain consumes one sample every two clocks, (initiation interval of 2), for the earlier example, the HDL domain needs to run 32 times faster than the AI Engine domain.

The following sections discuss setting the block parameters of the AIE to HDL and HDL to AIE blocks and includes examples.

AIE to HDL

The AIE to HDL block connects the output of AI Engine block with the input of HDL block. This block accepts variable size signal from AI Engine blocks along with the 'tready' signal which indicates the consumer can accept the data. The input data type to this block is inherited from the input signal.

Figure 282: **AIE to HDL**

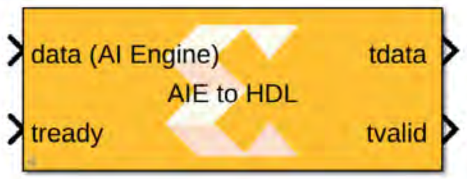
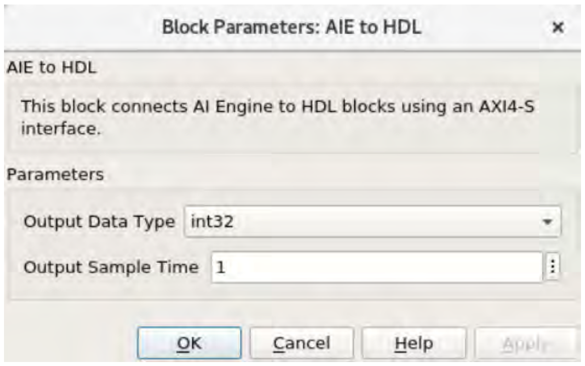


Figure 283: **AIE to HDL Parameters**


Block Parameters: AIE to HDL

AIE to HDL

This block connects AI Engine to HDL blocks using an AXI4-S interface.

Parameters

Output Data Type: int32

Output Sample Time: 1

Setting the AIE-HDL Block

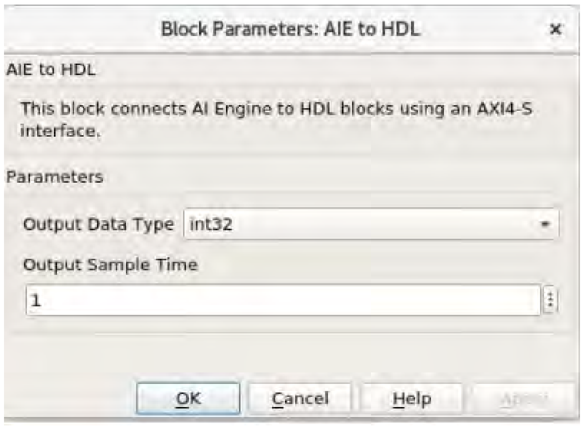
Step 1: Know the Initiation Interval (ii) of your HDL Design

A factor in setting the Output Sample Time in the AIE to HDL block is the initiation interval (II) of the HDL subsystem. As mentioned previously, simulation in HDL domain is cycle-accurate. An HDL design may not be ready to accept new data at every cycle (the `tready` signal from the HDL design will be set to zero when the HDL design cannot accept a new sample). For example, if an HDL design accepts a new sample every 10th cycle, the design is said to have an II of 10.

Step 2: Set the Parameter Output Data Type of the AIE to HDL Block

The `Output Data Type` parameter is limited to 32, 64, and 128 bits wide. This reflects the permissible data bit-width between AI Engine array and PL. There are more constraints in place. For example, if the input signal is of type `int64`, the output data type can only be of type `int64`. If the input is of type `int16(c)`, then the output should be `uint32`. Note that if you are using an AIE Signal Spec block to specify the PLIO width (to optimize throughput between AI Engine array and PL), then the Output Data Type should have the same number of bits as the PLIO width specified. In the absence of the AIE Signal Spec block, the generated code will have a PLIO width equal to the bitwidth of the signal leaving the AI Engine subsystem or 32 bits, whichever is larger. See Figure 286: AIE Kernel / AIE to HDL Block.

Figure 284: **AIE to HDL**



AIE to HDL

This block connects AI Engine to HDL blocks using an AXI4-S interface.

Parameters

Output Data Type: int32

Output Sample Time: 1

To get the list of Output data types that are supported by the AIE to HDL block and the corresponding input data type to the block, refer to the AIE to HDL block.

Following are some examples.

Data Type into the Block	PLIO	Output Data Type
int16	64 bits	uint64
int16(c)	Not set	uint32
int32	128 bits	ufix128
int8	Not set	uint32

Step 3 : Set the Parameter Output Sample Time of the AIE to HDL Block

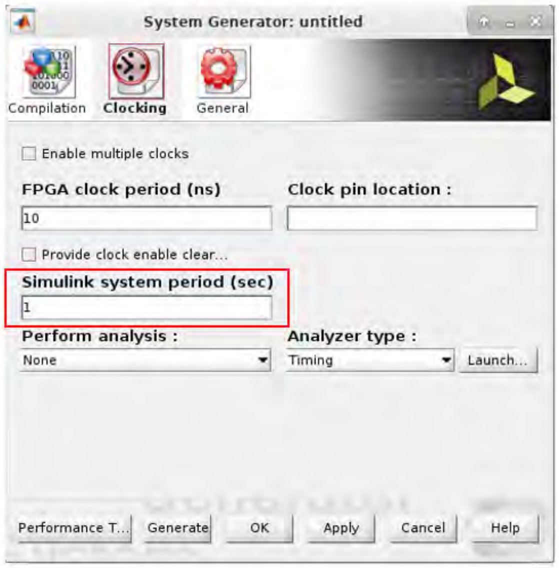
Set the `Output Sample Time` to (Input Sample Period)/(Input Size)/ii.

To understand the reason for this formula, assume ii is one (`tready` is always set to one). If the input to the AIE to HDL block is a variable-size signal of size *Input Size*, and the period is *Input Period* (you can determine the sample period by opening the Timing Legend in Simulink), this means in the time period *Input Period*, we are feeding *Input Size* samples into the block. To prevent the internal buffer of the block from overflowing, the output rate from the AIE to HDL block should be the same as its input. The input rate is (Input Size)/(Input Period) and the output rate is 1/(Output Sample Time). When ii is larger than one, the output rate is reduced to *1/(Output Sample Time)/ii*.

Step 4 : Set the System Clock

Open the Clocking tab in the System Generator block.There are two parameters there, the *FPGA clock period* and the *Simulink system period*. These two numbers define the scaling factor between time in a Simulink simulation, and time in the actual hardware implementation. Set the *Simulink system period* to the time calculated in step 3. Here it is assumed that the HDL design is a single-rate design. To learn more about these parameters, refer to the System Generator block.

Figure 285: **System Generator Clock Settings**


System Generator: untitled

Compilation | Clocking | General

☐ Enable multiple clocks

FPGA clock period (ns): 10

Clock pin location :

☐ Provide clock enable clear...

Simulink system period (sec): 1

Perform analysis : None

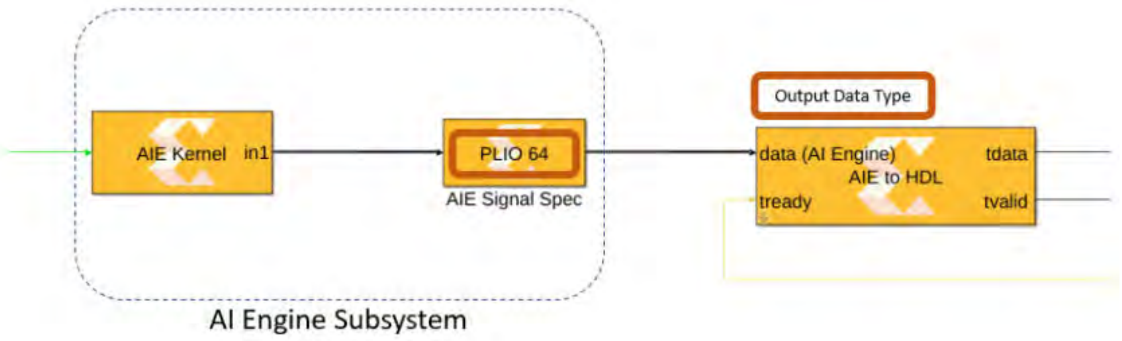
Analyzer type : Timing

Launch...

Performance T... | Generate | OK | Apply | Cancel | Help

Following is an example for connecting an AIE Kernel to an AIE to HDL block. The output datatype bit width should match the size of PLIO. Note that in absence of the PLIO block, the PLIO width is set to the larger signal bitwidth or 32-bit. With the PLIO block, you can specify a PLIO larger than 32-bits to improve throughput.

Figure 286: **AIE Kernel / AIE to HDL Block**



HDL to AIE

The HDL to AIE block connects the output of the HDL block with the input of the AI Engine block. This block accepts `tdata` which is the primary input for the data and the `tvalid` signal that indicates the producer has valid data. Output from the HDL to AIE block is a variable size signal (data) to AI Engine blocks along with the `tready` signal which indicates that the block can accept a transfer. A transfer takes place when both `tvalid` and `tready` are asserted.

Figure 287: **HDL to AIE**

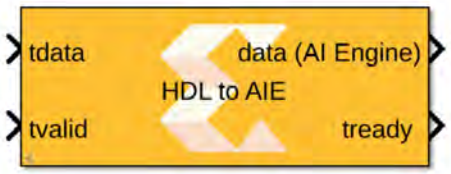
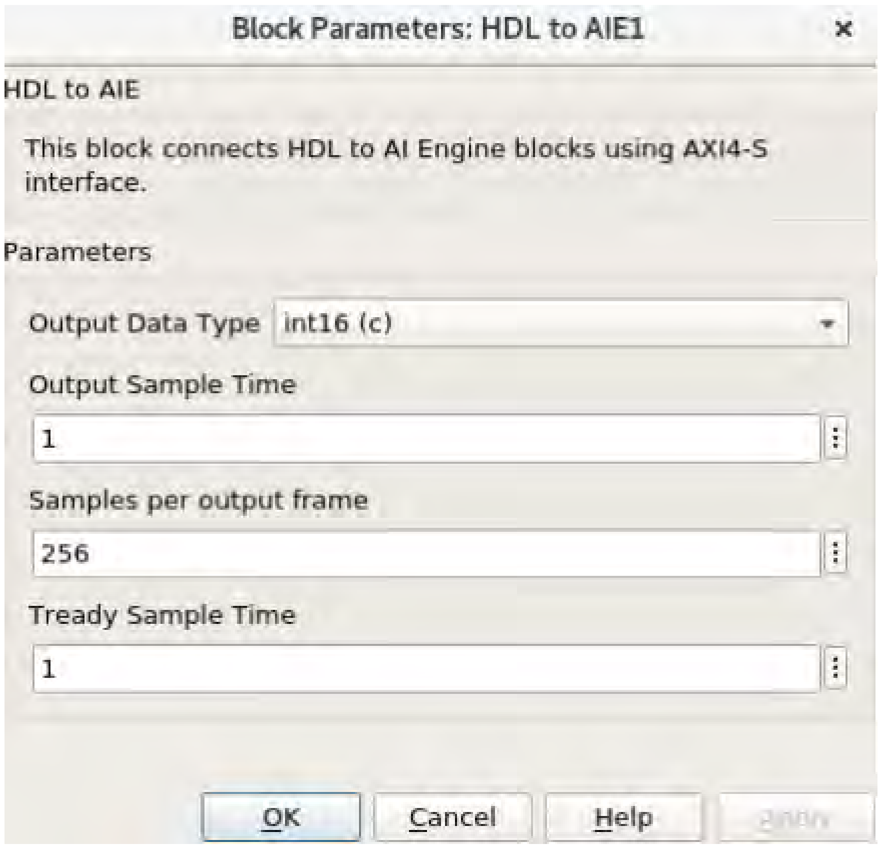


Figure 288: **HDL to AIE Parameters**



Block Parameters: HDL to AIE1

HDL to AIE

This block connects HDL to AI Engine blocks using AXI4-S interface.

Parameters

Output Data Type: int16 (c)

Output Sample Time: 1

Samples per output frame: 256

Tready Sample Time: 1

Setting the HDL-AIE Block

Step 1: Set the Output Data Type

The Output Data Type should be set to the data type that the consuming AI Engine block accepts. Note that the size you set for the PLIO should match the input bitwidth to the HDL to AIE block while the output data type of the HDL to AIE block should match the input data type of the consuming AIE block. See Figure 289: HDL-AIE to AIE Kernel Connection.

To get list of Output data types that are supported by the HDL to AIE block and the corresponding input data type to the block, refer to HDL to AIE.

The following table shows the Output Data Types that are supported by the HDL to AIE block and the corresponding Input data types to the block.

Step 2: Set the Output Frame Size

Assume that the consuming AIE block has a window input size of P, or it has a stream input that needs to read P samples to unblock (for example a `readincr_v4` requires four input samples to unblock). Set samples per output frame to P.

Step 3: Set the Output Sample Time

Set the Output Sample Time to:

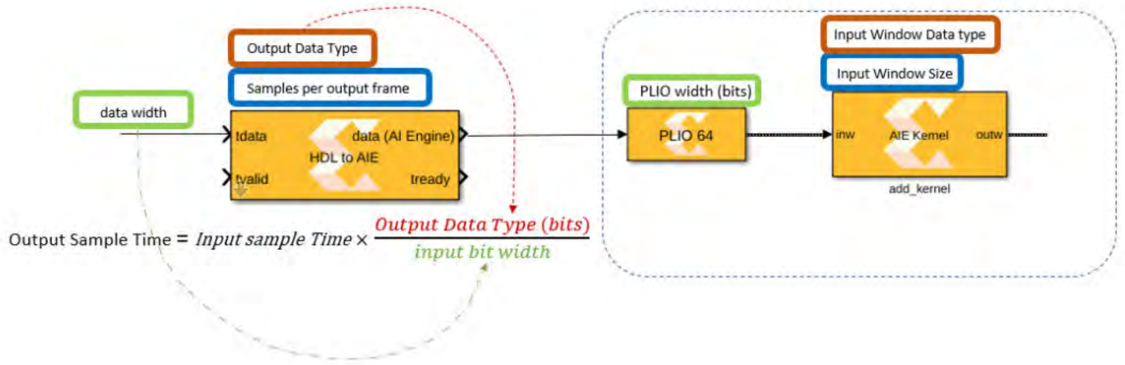
$$output\ sample\ time = input\ sample\ time * \frac{output\ bit\ width}{input\ bit\ width}$$

Step 4: Set the Tready Sample Time

The Tready Sample Time should be the same as the HDL design sample time.

The following figure shows an example for connecting the HDL to AIE block to an AIE Kernel with input window. Matching color boxes have the same values. Note that in the absence of a PLIO block the PLIO width is set to the larger of the signal bit width out of the HDL-AIE block and 32.

Figure 289: **HDL-AIE to AIE Kernel Connection**



Interconnecting AI Engines and HLS Kernels

HLS Function versus HLS Kernel

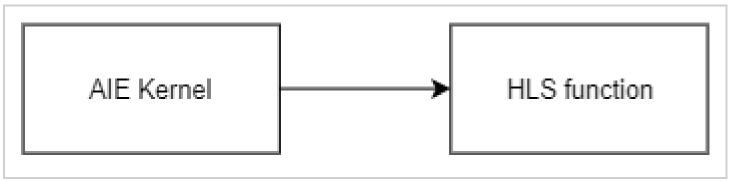
You can import the HLS function into your Model Composer design using the `xmcImportFunction` command as described in Importing C/C++ Code as Custom Blocks. You can simulate the block along with other blocks available in the Model Composer HLS library and generate HLS code from a system comprised of one or more imported HLS function blocks. Model Composer can also import HLS kernels. This section describes HLS Kernels and how it is different from an HLS function.

Behavior of HLS Functions on Blocking Calls

An HLS function first and foremost is a function. It has a predetermined number of inputs and outputs and every time the function is invoked, it consumes the inputs and produces the predetermined number of outputs. If an HLS function imported using `xmcImportFunction` hangs, (for example, if it has an infinite loop), Simulink will also hang, waiting indefinitely for the output from the imported block. This is because an imported HLS function using `xmcImportFunction` runs on the same thread as Simulink. If the imported functions hangs, Simulink also hangs.

While a function with an infinite loop is a rather trivial example, as a more practical example, assume you have an AI Engine kernel producing data and an HLS function consuming the data (see the following figure).

Figure 290: **Data Producer/Consumer**



The HLS function may have a blocking call reading the input. Following is a snippet of pseudo code outlining the HLS function

Pseudo code - Blocking call

```
void hls_func(hls::stream<ap_axis<64, 0, 0, 0> > & in_sample
              hls::stream<ap_axis<64, 0, 0, 0> > & out_sample) {
    ...
    ap_axis<64, 0, 0, 0> in_sample_x = in_sample.read();
    ...
}
```

In the previous code, `read()` is a blocking call. If there is no data on the stream, this call will block. The producing AI Engine kernel may or may not produce any output when invoked. As such, if Simulink calls the HLS Function with no data available from the producing block, the HLS function will block, and as a result Simulink will hang.

Note: In its current form, `xmcImportFunction` cannot import a function with a signature that includes `hls::stream`. For that, a wrapper is required.

Note: You cannot connect a block created using `xmcImportFucntion` with an AI Engine block as it does not accept the variable sized signals produced by AI Engine blocks.

Unlike an imported HLS function, an HLS Kernel block runs on a separate thread. As such, even if the HLS Kernel blocks (for example when waiting for input from the producing AI Engine block), Simulink will continue to function. In such cases, the output of the HLS kernel block will be a variable size signal containing no data.

HLS Kernels are IPs

When you import an HLS function into a design by itself, the HLS function will not operate as an IP with streaming ports. In Model Composer, you need to use the interface specification block to designate streaming ports for the design, and then generate the HLS IP. Unlike an HLS function, an HLS Kernel is a proper HLS IP that can be used in the Vitis™ software platform HLS and be synthesized directly. The following code snippet highlights the HLS kernel code with streaming interface.

`hls_kernel.cc`

```
void hls_kernel_blk(
    hls::stream<ap_axis<64, 0, 0, 0> > & in_sample1,
    hls::stream<ap_axis<64, 0, 0, 0> > & in_sample2,
    hls::stream<ap_axis<64, 0, 0, 0> > & out0_itr1,
    hls::stream<ap_axis<64, 0, 0, 0> > & out1_itr1
)
{
    #pragma HLS PIPELINE II=1
    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE axis register both port=out1_itr1
    #pragma HLS INTERFACE axis register both port=out0_itr1
    #pragma HLS INTERFACE axis register both port=in_sample1
    #pragma HLS INTERFACE axis register both port=in_sample2
    ap_int64 in_samp0 ; // Iteration-1: 2 complex samples concatenated to 
64-bit
    ap_int64 in_samp1 ; // Iteration-2: 2 complex samples concatenated to 
64-bit
...
```

In this example, notice the function signature and also the HLS pragmas specifying the interface on the ports. This function has all the constructs required by the HLS IP.

It is necessary to declare the corresponding kernel function in a specified format in the header file as follows.

```
void hls_kernel_blk(
    adf::dir::in hls::stream<ap_axis<64, 0, 0, 0> > &in_sample1,
    adf::dir::in hls::stream<ap_axis<64, 0, 0, 0> > &in_sample2,
    adf::dir::out hls::stream<ap_axis<64, 0, 0, 0> > &out0_itr1,
    adf::dir::out hls::stream<ap_axis<64, 0, 0, 0> > & out1_itr1
);
```

As shown, it is required to prepend each parameter in the function definition with either `adf::dir::in` or `adf::dir::out` based on the port direction.

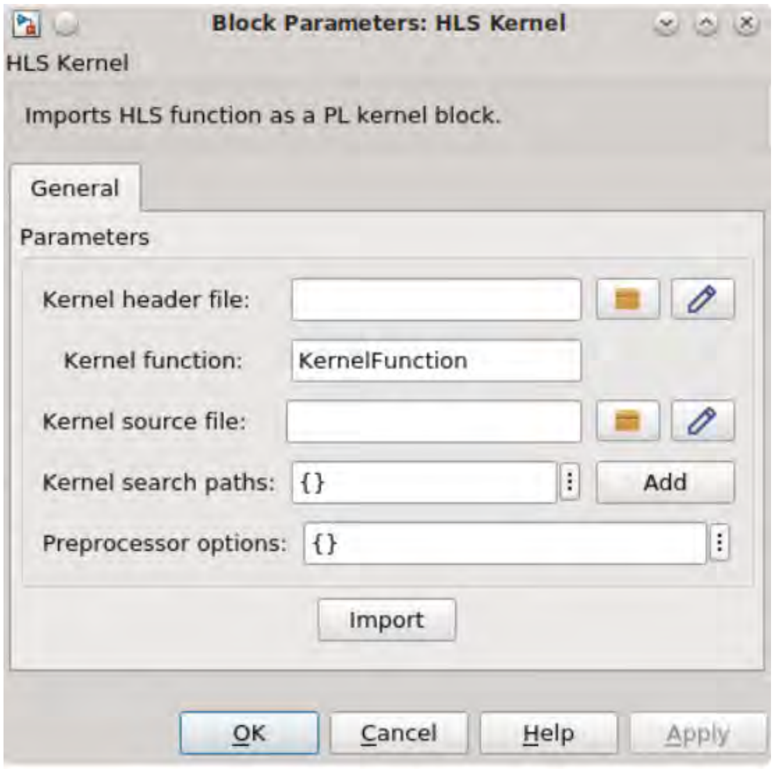
Importing HLS Kernels

To import the HLS kernel as a block into Model composer, you need to select it from the AI Engine library.

Figure 291: **HLS Kernel**

Double-click the block symbol to display the parameters of the HLS kernel block as shown in the following figure.

Figure 292: **HLS Kernel Parameters**


Block Parameters: HLS Kernel

HLS Kernel

Imports HLS function as a PL kernel block.

General

Parameters

Kernel header file:

Kernel function: KernelFunction

Kernel source file:

Kernel search paths: {}

Preprocessor options: {}

The block mask parameters need to be updated in order to import the HLS kernel as a block. The following table provides details on the parameters and descriptions for each parameter.

Table 28: **Parameters**

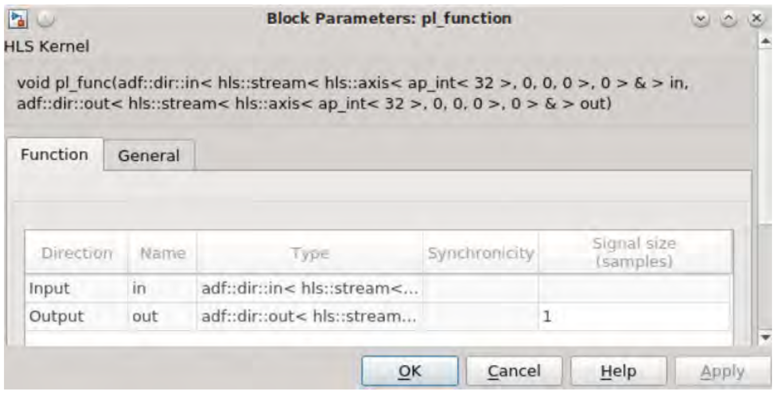
Parameter Name	Parameter Type	Criticality	Description
Kernel header file	String	Mandatory	The name of the HLS kernel header file that contains the function declaration. The string could be just the file name, a relative path to the file, or an absolute path of the file. Use the **Browse** button to select the file. If environment variables are used to specify the header file path, then an appropriate error is returned.
Kernel function	String	Mandatory	The name of the kernel function in C/C++ for which the HLS kernel block is to be created.
Kernel source file	String	Mandatory	The name of the source file that contains the kernel function implementation (definition). The string could be just the file name, a relative path to the file, or an absolute path of the file. If environment variables are used to specify the source file path, then an appropriate error is returned. Specifies the search path for source files (`.cc`, `.hpp`) from the MATLAB current folder.

Table 28: **Parameters** *(cont'd)*

Parameter Name	Parameter Type	Criticality	Description
Kernel search paths	Vector of Strings	Optional	If the kernel header file or the kernel source file is not found using the value provided through the `Kernel header file` or `Kernel source file` fields respectively, then the paths provided in `Kernel search paths` are used to locate the files. This parameter allows use of environment variables while specifying paths for the kernel header file and the kernel source file. The environment variable can be used in either `${ENV}` or `$ENV` format.
Preprocessor options		Optional	Optional preprocessor arguments for downstream compilation with specific preprocessor options. The following two preprocessor option formats will be accepted (multiple can be selected): `-Dname` and `-Dname=definition`. That is, the optional argument must begin with the `-D` string and if the option definition value is not provided, it is assumed to be `1`.

After successful import, the Function tab GUI displays automatically. You can quickly review the HLS Kernel definition and ports as shown in the following figure.

Figure 293: **Kernel Definition and Ports**



void pl_func(adf::dir::in< hls::stream< hls::axis< ap_int< 32 >, 0, 0, 0 >, 0 > & > in, adf::dir::out< hls::stream< hls::axis< ap_int< 32 >, 0, 0, 0 >, 0 > & > out)

Direction	Name	Type	Synchronicity	Signal size (samples)
Input	in	adf::dir::in< hls::stream<...		
Output	out	adf::dir::out< hls::stream...		1

Interconnect AI Engine and HLS Kernel Blocks

Connections between an output port of an AI Engine kernel and an input port of an HLS kernel use an AIE to HLS kernel block. Connections between an output port of an HLS kernel and an input port of an AI Engine kernel use an HLS to AIE kernel block. These blocks reformat the data to match the data type of the sink port. In this process no data (information) is lost; and it is simply adjusting the data type and the number of samples. For example, an interface block can reformat a signal carrying 64 `int8` values to a signal carrying 16 `int32` values.

These blocks are available in the `Xilinx Toolbox/AI Engines/Interfaces` library.

AIE to HLS Kernel

The AIE to HLS Kernel block reformats a signal driven by an AI Engine Kernel block or an AI Engine graph block, so that the resulting signal matches the data type and complexity required by the input of an HLS kernel block. For example, if the AI Engine kernel output port is of type `cint16` and the HLS kernel is of type `ap_axis<64>`, you would use an AIE to HLS kernel block with parameter output type set to `ap_axis<64>`. This block reads `cint16` samples from its input and then packs pairs of subsequent `cint16` samples into `ap_axis<64>` samples to its output.

Figure 294: **AIE to HLS Kernel Block**

Double-click the block symbol to see the parameters of the AIE to HLS kernel block. For more information refer to AIE to HLS.

- **Output Type:** Possible values are: ap_axis<32>, ap_axis<64>, ap_axis<128>, ap_axiu<32>, ap_axiu<64>, ap_axiu<128>, ap_int<32>, ap_int<64>, ap_uint<32>, ap_uint<64>,int, long long, unsigned, unsigned long long.
- **Output Size:** The size of the output port. The output port is a variable sized signal whose maximum size is specified by the OutputSize parameter. Default Output Size is `1`.

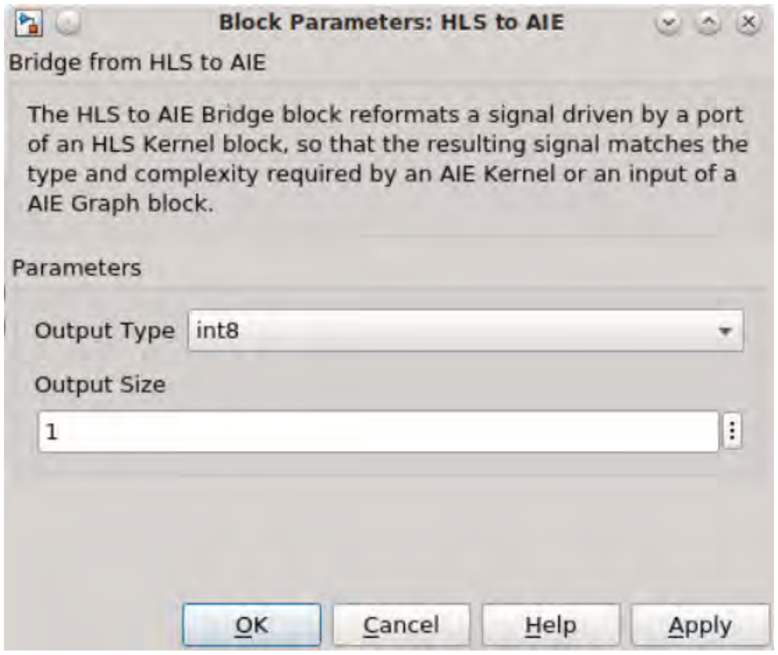
HLS Kernel to AIE

The HLS Kernel to AIE block reformats a signal driven by a port of an HLS kernel block, so that the resulting signal matches the data type and complexity required by an AI Engine kernel or an input of a AI Engine graph block. For example, if the data type of port of the HLS kernel block is `axiu<128>` and the data type of the port of the AI Engine is `uint32`, the HLS Kernel to AIE block reformats the input samples by unpacking each `axisu<128>` sample into four `uint32` samples. The output port of this block is a variable-size signal.

Figure 295: **HLS Kernel to AIE Block**

Double-click the block symbol to see the parameters of the HLS Kernel to AIE block. For more information refer to HLS to AIE.

Figure 296: **Block Parameters: HLS Kernel to AIE**



Block Parameters: HLS to AIE

Bridge from HLS to AIE

The HLS to AIE Bridge block reformats a signal driven by a port of an HLS Kernel block, so that the resulting signal matches the type and complexity required by an AIE Kernel or an input of a AIE Graph block.

Parameters

Output Type: int8

Output Size: 1

- **Output Type:** Possible values are: int8, int16, int32, int64, uint8, uint16, uint32, uint64, cint16, cint32.
- **Output Size:** The size of the output port. The output port is a variable-sized signal whose maximum size is specified by the Output Size parameter. Default size is '1'.

Connecting Source and Sink Blocks

The AI Engine library is compatible with the standard Simulink block library, and these blocks can be used together to create models that can be simulated in Simulink. However, only certain blocks which are designed to probe at different points in the design and debug are permitted inside a subsystem during code generation. Namely, Scope, Display, Spectrum Analyzer, To Workspace blocks etc. In addition to these, the AI Engine library provides some sink blocks that can be connected to the variable-size signal output from the AIE Kernel or AIE Graph blocks.

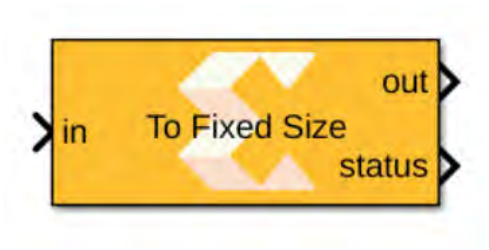
Block	Description
To Fixed Size	Converts variable-size output to fixed size.
Variable Size Signal to Workspace	Logs the variable signal to workspace.

To Fixed Size

The output ports of the AIE Kernel and AIE Graph blocks are variable-sized (vector) signals. There is a possibility that the kernel does not produce a fixed number of output samples in each simulation step. Many Simulink blocks do not accept variable-size signals as inputs and so this limits leveraging Simulink blocks in designs that use the AIE kernel and AIE Graph blocks.

Model Composer provides the To Fixed Size block which takes a variable-sized vector input and produces a fixed sized vector output.

Figure 297: **To Fixed Size**



The output vector size is specified by the Output Size parameter. The block copies samples from the input to the output. Excess samples are discarded. In cases where the input does not have enough samples, value 0 is used. The optional `status` output shows the difference between the number of samples in the input and output. The default value of the Output Size parameter is `1`. This block supports all the data types that are supported by Model Composer and the input can be real or complex.

Variable Size Signal to Workspace

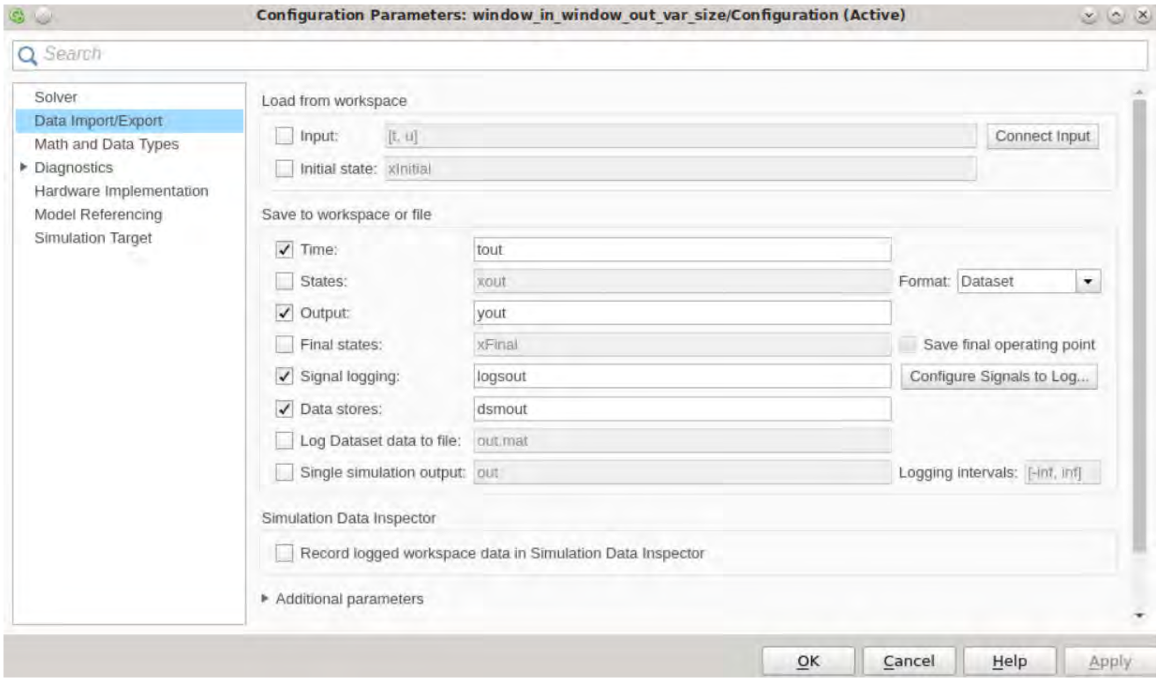
The output ports of AIE Kernel and AIE Graph blocks are variable-sized signals. Model Composer provides a Variable Size Signal to Workspace block in the AI Engine library to easily save the output into a workspace variable in MATLAB. In effect, this block is a mask on top of the Simulink To Workspace block.

Figure 298: **simout**

Within the block parameters, the default name for 'Variable name' is set to 'simout'.

Because the Variable Size Signal to Workspace block is using the Simulink To Workspace block, settings that change the behavior of the To Workspace block will impact the Variable Size Signal to Workspace block as well. The block settings can be accessed from Model Settings (Ctrl-E).

Figure 299: **Configuration Parameters**



Configuration Parameters: window_in_window_out_var_size/Configuration (Active)

Search

- Solver
- Data Import/Export
- Math and Data Types
- Diagnostics
- Hardware Implementation
- Model Referencing
- Simulation Target

Load from workspace

[ ] Input:	[t, u]	Connect Input
[ ] Initial state:	xInitial	

Save to workspace or file

[✓] Time:	tout		
[ ] States:	xout	Format:	Dataset
[✓] Output:	yout		
[ ] Final states:	xFinal	[ ] Save final operating point	
[✓] Signal logging:	logsout	Configure Signals to Log...	
[✓] Data stores:	dsmout		
[ ] Log Dataset data to file:	out.mat		
[ ] Single simulation output:	out	Logging intervals:	[-inf, inf]

Simulation Data Inspector

[ ] Record logged workspace data in Simulation Data Inspector

Additional parameters

OK Cancel Help Apply

You can set the name of the output (default is `out`) from the settings window and you can also control whether the To Workspace block outputs the results in a structure called `out`. Regardless of whether the checkbox is selected or what name is chosen, this block will work in a similar way as the To workspace block.

If you toggle the Single simulation output check box from Model Settings, press Ctrl-D to refresh the text the text on the Variable size signal to workspace block.

IMPORTANT! *If there is an error in the simulation, the block creates an empty variable when Single simulation output is not checked from the Model Settings window. This behavior is consistent with the To Workspace block.*

Connecting the AI Engine block to source blocks that generate or import signal data (Constant, Signal to Workspace block etc.) is supported.

Chapter 6

Xilinx Toolbox

Xilinx Toolbox Block Description

AI Engine Blocksets

Table 29: **AI Engine**

Block	Description
AIE to HDL	Connect the input port of an HDL block with the output port of an AI Engine kernel or AI Engine graph block using an AXI4-Stream interface.
HDL to AIE	Connect the output ports of HDL blocks to the input ports of AI Engine blocks using the AXI4-Stream protocol.
AIE to HLS	Connect an input port of an HLS kernel block to the output port of an AI Engine block in cases where the data type or complexity of the ports involved do not match.
HLS to AIE	Connect an input port of an AI Engine Kernel or AI Engine Graph block to an output port of an HLS Kernel block in cases where the datatype or complexities of the ports involved does not match.
AIE Signal Spec	Specify various properties on signals within, as well as at the boundary of an AI Engine subsystem.
To Fixed Size	Takes a variable size vector as an input and produces a fixed size vector as output.
Variable Size Signal to Workspace	Save variable size signal data from your Simulink® simulation to the MATLAB® workspace.
AIE Class Kernel	Import class-based AI Engine kernels
AIE Graph	Import an AI Engine graph.
AIE Kernel	Import an AI Engine kernel.
DDS	Implement the Direct Digital Synthesizer (DDS) targeted for AI Engines.
HLS Kernel	Import an HLS kernel code with a streaming interface.
FIR Asymmetric Decimation	Implements the FIR Asymmetric Decimation filter targeted for AI Engines.
FIR Asymmetric Filter	Implements the Single Rate Asymmetric FIR Filter targeted for AI Engines.
FIR Fractional Interpolation	Implements the FIR Fractional Asymmetric Interpolation filter targeted for AI Engines.
FIR Halfband Decimator	Implements the FIR Halfband Decimator targeted for AI Engines.
FIR Halfband Interpolator	Implements the FIR Halfband Interpolator targeted for AI Engines.
FIR Interpolation	Implements the FIR Asymmetric Interpolation filter targeted for AI Engines.

Table 29: **AI Engine** *(cont'd)*

Block	Description
FIR Symmetric Decimation	Implements the FIR Symmetric Decimation Filter targeted for AI Engines.
FIR Symmetric Filter	Implements the Single Rate Symmetric FIR Filter targeted for AI Engines.
IFFT	Implements the Inverse FFT targeted for AI Engines which use the rounding method and saturates the output samples on overflow.
FFT	Implements the FFT targeted for AI Engines which use rounding method and saturates the output samples on overflow.
Mixer	Implement the Mixer targeted for AI Engines.
RTP Source	Used as a source for the RTP input of an AI Engine block. When the RTP input is a scalar, the 'RTP Value' parameter should be a row vector. At each time step, the output is set to one of the elements of the vector starting with the first element. If an element of the vector is NaN, at the corresponding sampling time, the output will be an empty variable size signal.
To Variable Size	Takes a fixed sized vector input and produces a variable sized vector output. The maximum size of the output vector is specified by the Output Size parameter. If there is not enough samples to pack the output, the output will be an empty variable size signal.

HDL Blocksets

Basic Element Blocks

Table 30: **Basic Element Blocks**

Library	Description
Absolute	The Xilinx Absolute block outputs the absolute value of the input.
Accumulator	The Xilinx Accumulator block implements an adder or subtractor-based scaling accumulator.
AddSub	The Xilinx AddSub block implements an adder/subtractor. The operation can be fixed (Addition or Subtraction) or changed dynamically under control of the sub mode signal.
CMult	The Xilinx CMult block implements a gain operator, with output equal to the product of its input by a constant value. This value can be a MATLAB expression that evaluates to a constant.
Convert	The Xilinx Convert block converts each input sample to a number of a desired arithmetic type. For example, a number can be converted to a signed (two's complement), or unsigned value.
Depuncture	The Xilinx Depuncture block allows you to insert an arbitrary symbol into your input data at the location specified by the depuncture code.
Divide	The Xilinx Divide block performs both fixed-point and floating-point division with the a input being the dividend and the b input the divisor. Both inputs must be of the same data type.
Down Sample	The Xilinx Down Sample block reduces the sample rate at the point where the block is placed in your design.
Exponential	This Xilinx Exponential block preforms the exponential operation on the input. Currently, only the floating-point data type is supported.
Expression	The Xilinx Expression block performs a bitwise logical expression.
Mult	The Xilinx Mult block implements a multiplier. It computes the product of the data on its two input ports, producing the result on its output port.

Table 30: **Basic Element Blocks** *(cont'd)*

Library	Description
MultAdd	The Xilinx MultAdd block performs both fixed-point and floating-point multiply and addition with the a and b inputs used for the multiplication and the c input for addition or subtraction.
Mux	The Xilinx Mux block implements a multiplexer. The block has one select input (type unsigned), and a user-configurable number of data bus inputs, ranging from 2 to 1024.
Natural Logarithm	The Xilinx Natural Logarithm block produces the natural logarithm of the input.
Negate	The Xilinx Negate block computes the arithmetic negation of its input.
Parallel to Serial	The Parallel to Serial block takes an input word and splits it into N time-multiplexed output words where N is the ratio of number of input bits to output bits. The order of the output can be either least significant bit first or most significant bit first.
Puncture	The Xilinx Puncture block removes a set of user-specified bits from the input words of its data stream.
Reciprocal	The Xilinx Reciprocal block performs the reciprocal on the input. Currently, only the floating-point data type is supported.
Reciprocal SquareRoot	The Xilinx Reciprocal SquareRoot block performs the reciprocal squareroot on the input. Currently, only the floating-point data type is supported.
Reinterpret	The Xilinx Reinterpret block forces its output to a new type without any regard for retaining the numerical value represented by the input.
Relational	The Xilinx Relational block implements a comparator.
Requantize	The Xilinx Requantize block requantizes and scales its input signals.
Scale	The Xilinx Scale block scales its input by a power of two. The power can be either positive or negative. The block has one input and one output. The scale operation has the effect of moving the binary point without changing the bits in the container.
Serial to Parallel	The Serial to Parallel block takes a series of inputs of any size and creates a single output of a specified multiple of that size. The input series can be ordered either with the most significant word first or the least significant word first.
Shift	The Xilinx Shift block performs a left or right shift on the input signal. The result will have the same fixed-point container as that of the input.
Slice	The Xilinx Slice block allows you to slice off a sequence of bits from your input data and create a new data value. This value is presented as the output from the block. The output data type is unsigned with its binary point at zero.
SquareRoot	The Xilinx SquareRoot block performs the square root on the input. Currently, only the floating-point data type is supported.
Threshold	The Xilinx Threshold block tests the sign of the input number. If the input number is negative, the output of the block is -1; otherwise, the output is 1. The output is a signed fixed-point integer that is 2 bits long. The block has one input and one output.
Time Division Demultiplexer	The Xilinx Time Division Demultiplexer block accepts input serially and presents it to multiple outputs at a slower rate.
Time Division Multiplexer	The Xilinx Time Division Multiplexer block multiplexes values presented at input ports into a single faster rate output stream.
Up Sample	The Xilinx Up Sample block increases the sample rate at the point where the block is placed in your design. The output sample period is l/n, where l is the input sample period, and n is the sampling rate.

DSP Blocks

Table 31: **DSP Blocks**

Library	Description
Digital FIR Filter	The Xilinx Digital FIR Filter block allows you to generate highly parameterizable, area-efficient, high-performance single channel FIR filters.
DSP Macro 1.0	The Xilinx DSP macro block provides a device independent abstraction of the DSP48E1, DSP48E2, and DSP58 blocks. Using this block instead of using a technology-specific DSP slice helps makes the design more portable between Xilinx technologies.
DSP48E	The Xilinx DSP48E block is an efficient building block for DSP applications that use supported devices. The DSP48E combines an 18-bit by 25-bit signed multiplier with a 48-bit adder and programmable mux to select the adder's input.
DSP48E1	The Xilinx DSP48E1 block is an efficient building block for DSP applications that use 7 series devices. Enhancements to the DSP48E1 slice provide improved flexibility and utilization, improved efficiency of applications, reduced overall power consumption, and increased maximum frequency. The high performance allows designers to implement multiple slower operations in a single DSP48E1 slice using time-multiplexing methods.
DSP48E2	The Xilinx DSP48E2 block is an efficient building block for DSP applications that use UltraScale™ devices. DSP applications use many binary multipliers and accumulators that are best implemented in dedicated DSP resources. UltraScale™ devices have many dedicated low-power DSP slices, combining high speed with small size while retaining system design flexibility.
DSP58	The Xilinx DSP58 block is an efficient building block for DSP applications that use Versal™ devices. DSP applications use many binary multipliers and accumulators that are best implemented in dedicated DSP resources. Versal™ devices have many dedicated low-power DSP slices, combining high speed with small size while retaining system design flexibility.
DSPCPLX	The Xilinx DSPCPLX block is one of the advanced features provided by Versal™ architecture DSP, which is the optimized solution to deal with 18x18 complex multiplication followed by 58 + 58 accumulation operation.
FFT	The Xilinx FFT (Fast Fourier Transform) block takes a block of time domain waveform data and computes the frequency of the sinusoid signals that make up the waveform.
Inverse FFT	The Xilinx Inverter FFT block performs a fast inverse (or backward) Fourier transform (IDFT), which undoes the process of Discrete Fourier Transform (DFT). The Inverter FFT maps the signal back from the frequency domain into the time domain.
Product	The Xilinx Product block implements a scalar or complex multiplier. It computes the product of the data on its two input channels, producing the result on its output channel. For complex multiplication the input and output have two components: real and imaginary.
Sine Wave	The Xilinx Sine Wave block generates a sine wave, using simulation time as the time source.
CIC Compiler 4.0	The Xilinx CIC Compiler provides the ability to design and implement AXI4-Stream-compliant Cascaded Integrator-Comb (CIC) filters for a variety of Xilinx FPGA devices.
Complex Multiplier 6.0	The Complex Multiplier block implements AXI4-Stream compliant, high-performance, optimized complex multipliers for devices based on user-specified options.
Convolution Encoder 9.0	The Xilinx Convolution Encoder block implements an encoder for convolutioncodes. Ordinarily used in tandem with a Viterbi decoder, this block performsforward error correction (FEC) in digital communication systems. This block adheres to the AMBA AXI4-Stream standard.

Table 31: **DSP Blocks** *(cont'd)*

Library	Description
CORDIC 6.0	The Xilinx CORDIC block implements a generalized coordinate rotational digital computer (CORDIC) algorithm and is AXI compliant.
DDS Compiler 6.0	The Xilinx DDS (Direct Digital Synthesizer) Compiler block implements high performance, optimized Phase Generation, and Phase to Sinusoid circuits with AXI4-Stream compliant interfaces for supported devices.
Divider Generator 5.1	The Xilinx Divider Generator block creates a circuit for integer division based on Radix-2 non-restoring division, or High-Radix division with prescaling.
Fast Fourier Transform 9.1	The Xilinx Fast Fourier Transform block implements the Cooley-Tukey FFT algorithm, a computationally efficient method for calculating the Discrete Fourier Transform (DFT). In addition, the block provides an AXI4-Stream-compliant interface.
FIR Compiler 7.2	This Xilinx FIR Compiler block provides users with a way to generate highly parameterizable, area-efficient, high-performance FIR filters with an AXI4-Stream-compliant interface.
Interleaver/De-interleaver 8.0	The Xilinx Interleaver Deinterleaver block implements an interleaver or a deinterleaver using an AXI4-compliant block interface. An interleaver is a device that rearranges the order of a sequence of input symbols. The term symbol is used to describe a collection of bits. In some applications, a symbol is a single bit. In others, a symbol is a bus.
Reed-Solomon Decoder 9.0	The Reed-Solomon (RS) codes are block-based error correcting codes with a wide range of applications in digital communications and storage.
Reed-Solomon Encoder 9.0	The Reed-Solomon (RS) codes are block-based error correcting codes with a wide range of applications in digital communications and storage. This block adheres to the AMBA® AXI4-Stream standard.
Viterbi Decoder 9.1	Data encoded with a convolution encoder can be decoded using the Xilinx Viterbi decoder block. This block adheres to the AMBA® AXI4-Stream standard.

Interface Blocks

Table 32: **Interface Blocks**

Library	Description
Gateway In	The Xilinx Gateway In blocks are the inputs into the Xilinx portion of your Simulink design. These blocks convert Simulink integer, double, and fixed-point data types into the Model Composer fixed-point type. Each block defines a top-level input port or interface in the HDL design generated by Model Composer.
Gateway Out	Xilinx Gateway Out blocks are the outputs from the Xilinx portion of your Simulink design. This block converts the Model Composer fixed-point or floating-point data type into a Simulink integer, single, double, or fixed-point data type.

Logic and Bit Operation Blocks

Table 33: **Logic and Bit Operation Blocks**

Library	Description
Assert	The Xilinx Assert block is used to assert a rate and/or a type on a signal. This block has no cost in hardware and can be used to resolve rates and/or types in situations where designer intervention is required.
BitBasher	The Xilinx BitBasher block performs slicing, concatenation, and augmentation of inputs attached to the block.

Table 33: **Logic and Bit Operation Blocks** *(cont'd)*

Library	Description
Concat	The Xilinx Concat block performs a concatenation of n bit vectors represented by unsigned integer numbers, for example, n unsigned numbers with binary points at position zero.
Inverter	The Xilinx Inverter block calculates the bitwise logical complement of a fixed-point number. The block is implemented as a synthesizable VHDL module.
Logical	The Xilinx Logical block performs bitwise logical operations on fixed-point numbers. Operands are zero padded and sign extended as necessary to make binary point positions coincide; then the logical operation is performed and the result is delivered at the output port.

Memory Blocks

Table 34: **Memory Blocks**

Memory Block	Description
Addressable Shift Register	The Xilinx Addressable Shift Register block is a variable-length shift register in which any register in the delay chain can be addressed and driven onto the output data port.
Delay	The Xilinx Delay block implements a fixed delay of L cycles.
Dual Port RAM	The Xilinx Dual Port RAM block implements a random access memory (RAM). Dual ports enable simultaneous access to the memory space at different sample rates using multiple data widths.
FIFO	The Xilinx FIFO block implements an FIFO memory queue.
LFSR	The Xilinx LFSR block implements a Linear Feedback Shift Register (LFSR). This block supports both the Galois and Fibonacci structures using either the XOR or XNOR gate and allows a re-loadable input to change the current value of the register at any time. The LFSR output and re-loadable input can be configured as either serial or parallel ports.
Register	The Xilinx Register block models a D flip-flop-based register, having a latency of one sample period.
ROM	The Xilinx ROM block is a single port read-only memory (ROM).
Single Port RAM	The Xilinx Single Port RAM block implements a random access memory (RAM) with one data input and one data output port.
AXI FIFO	The Xilinx AXI FIFO block implements a FIFO memory queue with an AXI-compatible block interface.

Signal Routing Blocks

Table 35: **Signal Routing Blocks**

Library	Description
Bus Creator	This block creates buses from input signals
Bus Selector	This block selects signals from incoming buses
From	This block accepts inputs from the Goto block
Goto	This block passes block inputs to theFrom blocks

Source Blocks

Table 36: **Source Blocks**

Library	Description
Constant	The Xilinx Constant block generates a constant that can be a fixed-point value, a Boolean value, or a DSP48 instruction. This block is similar to the Simulink constant block, but can be used to directly drive the inputs on Xilinx blocks.
Counter	The Xilinx Counter block implements a free-running or count-limited type of an up, down, or up/down counter. The counter output can be specified as a signed or unsigned fixed-point number.
Opmode	The Xilinx Opmode block generates a constant that is a DSP48E, DSP48E1, or DSP48E2 instruction. It is is a 15-bit instruction for DSP48E, a 20-bit instruction for DSP48E1, and a 22-bit instruction for DSP48E2. The instruction consists of the opmode, carry-in, carry-in select, alumode, and (for DSP48E1 and DSP48E2) the inmode bits.
Reset Generator	The Xilinx Reset Generator block captures the user's reset signal that is running at the system samplerate, and produces one or more downsampled reset signal(s) running at the rates specified on the block.

SSR Blocks

Table 37: **SSR Blocks**

Library	Description
Vector Absolute	The Vector Absolute block outputs the absolute value of the input of vector type.
Vector AddSub Fabric	The Vector Adder/Subtracter Fabric block supports the Addition/Subtraction operation forinputs of vector type.
Vector Assert	The Vector Assert block asserts a user-defined sample rate and/or type on Vector inputs.
Vector Concat	The Vector Concat block concatenates two or more inputs of type vector. The output is cast toan unsigned value with the binary point at zero.
Vector Convert	The Vector Convert block supports Data Type Conversion feature for vector type inputs.
Vector Down Sample	The Vector Down Sample block down samples input vector data.
Vector Logical	The Vector Logical block supports logical operation for vector type inputs.
Vector Mux	The Vector Multiplexer block supports the Multiplexing feature for input of vector types.
Vector Real Mult	The Vector Real Multiplier block supports the multiplication feature for vector type inputs.
Vector Reinterpret	The Vector Reinterpret block changes the vector input signal type without altering the binary representation.
Vector Relational	The Vector Relational block implements comparator for vector inputs.
Vector Slice	The Vector Slice block extracts a given range of bits from each sample of input vector and presents it at the output.
Vector Up Sample	The Vector Up Sample block up samples input vector data. Inserted values can be zeros or copies of the most recent input sample.
Vector Complex Mult	The Vector Complex Multiplier block supports multiplication of two complex input vectors.
Vector DDFS	The Vector DDFS block generates Real and Imaginary vector output signals of desired frequency.

Table 37: **SSR Blocks** *(cont'd)*

Library	Description
Vector FFT	The Vector FFT block supports the FFT operation for vector type inputs.
Vector FIR	The Vector FIR block supports FIR filtering for vector type inputs.
Scalar2Vector	The Scalar2Vector block converts scalar type input to vector type output.
Vector Real Gateway In	The Vector Real Gateway In block converts vector inputs of type Simulink® integer, single,double, and fixed-point to Xilinx® fixed-point or floating-point data type.
Vector Real Gateway Out	The Vector Real Gateway Out block converts Xilinx® fixed-point or floating-point type vectorinputs into vector outputs of type Simulink® integer, single, double, or fixed-point.
Vector2Scalar	The Vector2Scalar block converts vector type input to scalar type output.
Vector Delay	The Vector Delay block supports delay operation on vector type inputs.
Vector Delay Delta	The Vector Delay Delta Block delays each vector element differently based on the given latencyand delay latency values.
Vector Register	The Vector Register block supports vector type inputs.
Vector Constant	The Vector Constant Block generates vector constant values.

Tools

Table 38: **Tools**

Library	Description
System Generator	The System Generator token serves as a control panel for controlling system and simulation parameters, and it is also used to invoke the code generator for netlisting. Every Simulink model containing any element from the HDL Blockset must contain at least one System Generator token. Once a System Generator token is added to a model, it is possible to specify how code generation and simulation should be handled.
Clock Enable Probe	The Xilinx Clock Enable (CE) Probe provides a mechanism for extracting derived clock enable signals from Xilinx signals in Model Composer models.
Clock Probe	The Xilinx Clock Probe generates a double-precision representation of a clock signal with a period equal to the Simulink system period.
FDATool	The Xilinx FDATool block provides an interface to the FDATool software available as part of the MATLAB Signal Processing Toolbox.
Indeterminate Probe	The output of the Xilinx Indeterminate Probe indicates whether the input data is indeterminate (MATLAB value NaN). An indeterminate data value corresponds to a VHDL indeterminate logic data value of 'X'.
Questa	The HDL Black Box block provides a way to incorporate existing HDL files into a model. When the model is simulated, co-simulation can be used to allow black boxes to participate. The Questa HDL co-simulation block configures and controls co-simulation for one or several black boxes.
Sample Time	The Sample Time block reports the normalized sample period of its input. A signal's normalized sample period is not equivalent to its Simulink absolute sample period. In hardware, this block is implemented as a constant.

User-Defined functions

Table 39: **User-Defined functions**

Library	Description
Black Box	The HDL Black Box block provides a way to incorporate hardware description language (HDL) models into Model Composer.
MCode	The Xilinx MCode block is a container for executing a user-supplied MATLAB® function within Simulink. A parameter on the block specifies the M-function name. The block executes the M-code to calculate block outputs during a Simulink simulation. The same code is translated in a straightforward way into equivalent behavioral VHDL/Verilog when hardware is generated.
Vitis HLS	The Xilinx Vitis™ HLS block allows the functionality of a Vitis HLS design to be included in a Model Composer design. The Vitis HLS design can include C, C++, and System C design sources.

HLS Blocksets

Table 40: **Logic and Bit Operations**

Block	Description
Bit Concat	Perform bitwise concatenation of input values into a single output value
Bit Slice	Extract a range of bits from a value
Bitwise AND	Perform element and bitwise Boolean AND operation on the inputs
Bitwise NOT	Perform element and bitwise Boolean NOT operation on the input
Bitwise OR	Perform element and bitwise Boolean OR operation on the inputs
Bitwise XOR	Perform element and bitwise Boolean XOR operation on the inputs
Logical AND	Performs element-wise logical AND operation on inputs
Logical NOT	Performs element-wise logical NOT operation on the input
Logical OR	Performs element-wise logical OR operation on inputs
Reduction AND	Compute bitwise AND of the elements of the input over all dimensions or over a specified dimension
Reduction OR	Compute bitwise OR of the elements of the input over all dimensions or over a specified dimension
Reduction XOR	Compute bitwise XOR of the elements of the input over all dimensions or over a specified dimension
Shift Left	Perform logical shift left of input over a constant number of bit positions specified by a non-negative integer mask parameter
Shift Right	Perform logical shift right of input over a constant number of bit positions specified by a non-negative integer mask parameter

Table 41: **Lookup Tables**

Block	Description
Lookup Table	Perform one-dimensional lookup operation with an input index

Table 42: **Math Functions / Math Operations**

Block	Description
Abs	Compute element-wise absolute value of input signal
atan	Compute element-wise inverse tangent of input signal
atan2	Compute element-wise four-quadrant inverse tangent of input signal
Complex to Polar	Element-wise conversion of complex input signals into magnitude and radiant phase angle
Complex to Real-Imag	Output real and imaginary parts of complex input signal
Conjugate	Apply element-wise complex conjugate operation to the input signal
Cosine	Element-wise computation of the cosine function for a given argument
cosh	Element-wise computation of the hyperbolic cosine for a given argument
Cumulative Sum	Compute the cumulative sum along the specified dimension of the input
Divide	Perform element-wise division
Exp	Perform an element-wise exponential value of the input
Gain	Multiply the input signal with a constant gain factor.
Log	Compute element-wise natural logarithm of input
Log10	Compute element wise base 10 logarithm of input
Max	Computes the maximum value of an input or element-wise maximum value of multiple inputs.
Min	Computes the minimum value of an input or element-wise minimum value of multiple inputs.
Modulus	Perform element-wise modulus operation on the input signals
Negate	Perform element-wise unary minus operation on the input data
Polar to Complex	Element-wise conversion of real magnitude and angle representation signals into a complex signal
Pow	Compute the element-wise power function
Product	Compute element-wise product of the input signals
Product of Elements	Multiply the elements of the input signal
Real-Imag to Complex	Convert real and/or imaginary inputs to complex signal
Reciprocal	Perform element-wise computation of the reciprocal for a given argument
Reciprocal Sqrt	Perform element-wise computation of the reciprocal square root for a given argument
Remainder	Perform element-wise division on the input signal, and the output is the remainder after the division
Reshape Row-Major	Changes the input dimensions in row-major order.
Signum	Perform an element-wise signum function (sign extraction)
Sine	Element-wise computation of the sine function for the given input
sinh	Element-wise computation of the hyperbolic sine for a given argument
Sqrt	Element-wise computation of the square root for a given argument
Subtract	Perform an element-wise subtraction
Sum	Performs element-wise addition of two input signals
Sum of Elements	Perform element-wise addition on the input, column-wise, row-wise, or in all-dimensions

Table 42: **Math Functions / Math Operations** *(cont'd)*

Block	Description
Tangent	Perform an element-wise computation of the tangent function for the given argument

Table 43: **Math Functions / Matrices and Linear Algebra**

Block	Description
Hermitian	Perform element-wise conjugate transpose operation on the input signal
Matrix Multiply	Compute matrix product of two input signals
QR Inverse	Compute the inverse of a matrix using QR factorization
Submatrix	Select a subset of elements (submatrix) from matrix input
Transpose	Perform an element-wise transpose operation on the input signal

Table 44: **Ports and Subsystems**

Block	Description
If	Model `if-else` control flow
In1	Create input port for subsystem or external input
Out1	Create output port for subsystem or external output
Action Port	Implement Action subsystems used in `if` and `switch` control flow statements
Window Processing	Assemble an output matrix by applying the kernel subsystem to submatrices (windows) of the input matrix in row-major order

Table 45: **Relational Operations**

Block	Description
Equals	Perform element-wise equal to relational operation on the inputs
Greater	Perform element-wise greater than relational operation on the inputs
Greater Equals	Perform element-wise greater than or equal relational operation on the inputs
Lesser	Perform element-wise less than relational operation on the inputs
Lesser Equals	Perform element-wise less than or equal relational operation on the inputs
Not Equals	Perform element-wise not equal to relational operation on the inputs

Table 46: **Signal Attributes**

Block	Description
Data Type Conversion	Convert the input to the data type of the output
Reinterpret	Element-wise reinterpretation of the input type into a compatible output type with the same bit width

Table 47: **Signal Operations**

Block	Description
Delay	Delay input signal by specified number of samples
Unit Delay	Provides a delay of one sample period

Table 48: **Signal Routing**

Block	Description
Bus Creator	Create Signal Bus
Bus Selector	Select signals from incoming bus
Conditional	Pass through input T when control input satisfies a selected criterion; otherwise, pass through input F
Demux	Separates a vector input into a number of scalar and vector outputs
From	Accept input from Goto block
Goto	Pass block input to From blocks
Merge	Combine multiple signals into single signal
Mux	Combines scalar and vector inputs into a larger vector output

Table 49: **Sinks**

Block	Description
Display	Show value of input
Scope	Display signals generated during simulation
Stop Simulation	Stop simulation when input is nonzero
Terminator	Terminate unconnected output port
To File	Write data to file
To Workspace	Write data to workspace

Table 50: **Source**

Block	Description
Constant	Generates the constant specified by the **Constant Value** parameter

Table 51: **Tools**

Block	Description
DocBlock	Create text that documents model and save text with model
Interface Spec	Specify the RTL interfaces for a subsystem
Model Composer Hub	Control implementation of the model

HDL Blockset

Common Options in Block Parameter Dialog Boxes

Each Xilinx® block has several controls and configurable parameters, seen in its block parameters dialog box. This dialog box can be accessed by double-clicking on the block. Many of these parameters are specific to the block. Block-specific parameters are described in the documentation for the block.

The remaining controls and parameters are common to most blocks. These common controls and parameters are described below.

Each dialog box contains four buttons: **OK**, **Cancel**, **Help**, and **Apply**. **Apply** applies configuration changes to the block, leaving the box open on the screen. **Help** displays HTML help for the block. **Cancel** closes the box without saving changes. **OK** applies changes and closes the box.

Precision

The fundamental computational mode in the Xilinx blockset is arbitrary precision fixed-point arithmetic. Most blocks give you the option of choosing the precision, for example, the number of bits and binary point position.

By default, the output of Xilinx blocks is *full* precision; that is, sufficient precision to represent the result without error. Most blocks have a *User-Defined* precision option that fixes the number of total and fractional bits.

Arithmetic Type

In the **Type** field of the block parameters dialog box, you can choose unsigned or signed (two's complement) as the data type of the output signal.

Number of Bits

Fixed-point numbers are stored in data types characterized by their word size as specified by number of bits, binary point, and arithmetic type parameters. The maximum number of bits supported is 4096.

Binary Point

The binary point is the means by which fixed-point numbers are scaled. The binary point parameter indicates the number of bits to the right of the binary point (for example, the size of the fraction) for the output port. The binary point position must be between zero and the specified number of bits.

Overflow and Quantization

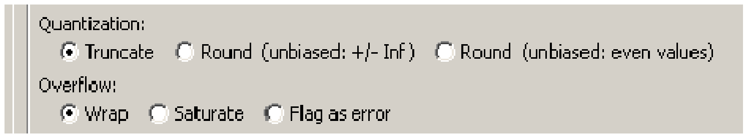
When user-defined precision is selected, errors can result from overflow or quantization. Overflow errors occur when a value lies outside the representable range. Quantization errors occur when the number of fractional bits is insufficient to represent the fractional portion of a value.

The Xilinx fixed-point data type supports several options for user-defined precision. For overflow the options are to **Saturate** to the largest positive/smallest negative value, to **Wrap** (for example, to discard bits to the left of the most significant representable bit), or to **Flag as error** (an overflow as a Simulink® error) during simulation. **Flag as error** is a simulation only feature. The hardware generated is the same as when **Wrap** is selected.

For quantization, the options are to **Round** to the nearest representable value (or to the value furthest from zero if there are two equidistant nearest representable values), or to **Truncate** (for example, to discard bits to the right of the least significant representable bit).

The following is an image showing the Quantization and Overflow options.

Figure 300: **Quantization and Overflow Options**



Quantization:
[✓ Truncate] Round (unbiased: +/- Inf) Round (unbiased: even values)
Overflow:
[✓ Wrap] Saturate Flag as error

Round(unbiased: +/- inf) also known as "Symmetric Round (towards +/- inf)" or "Symmetric Round (away from zero)". This is similar to the MATLAB® `round()` function. This method rounds the value to the nearest desired bit away from zero and when there is a value at the midpoint between two possible rounded values, the one with the larger magnitude is selected. For example, to round 01.0110 to a Fix_4_2, this yields 01.10, because 01.0110 is exactly between 01.01 and 01.10 and the latter is further from zero.

Round (unbiased: even values) also known as "Convergent Round (toward even)" or "Unbiased Rounding". Symmetric rounding is biased because it rounds all ambiguous midpoints away from zero which means the average magnitude of the rounded results is larger than the average magnitude of the raw results. Convergent rounding removes this by alternating between a symmetric round toward zero and symmetric round away from zero. That is, midpoints are rounded toward the nearest even number. For example, to round 01.0110 to a Fix_4_2, this yields 01.10, because 01.0110 is exactly between 01.01 and 01.10 and the latter is even. To round 01.1010 to a Fix_4_2, this yields 01.10, because 01.1010 is exactly between 01.10 and 01.11 and the former is even.

It is important to realize that whatever option is selected, the generated HDL model and Simulink model behave identically.

Latency

Many elements in the Xilinx blockset have a latency option. This defines the number of sample periods by which the block's output is delayed. One sample period might correspond to multiple clock cycles in the corresponding FPGA implementation (for example, when the hardware is over-clocked with respect to the Simulink model). Model Composer does not perform extensive pipelining; additional latency is usually implemented as a shift register on the output of the block.

Provide Synchronous Reset Port

Selecting the **Provide Synchronous Reset Port** option activates an optional reset (rst) pin on the block.

When the reset signal is asserted the block goes back to its initial state. Reset signal has precedence over the optional enable signal available on the block. The reset signal has to run at a multiple of the block's sample rate. The signal driving the reset port must be Boolean.

Provide Enable Port

Selecting the **Provide Enable Port** option activates an optional enable (en) pin on the block. When the enable signal is not asserted the block holds its current state until the enable signal is asserted again or the reset signal is asserted. Reset signal has precedence over the enable signal. The enable signal has to run at a multiple of the block 's sample rate. The signal driving the enable port must be Boolean.

Sample Period

Data streams are processed at a specific sample rate as they flow through Simulink. Typically, each block detects the input sample rate and produces the correct sample rate on its output. Xilinx blocks Up Sample and Down Sample provide a means to increase or decrease sample rates.

Specify Explicit Sample Period

If you select **Specify explicit sample period** rather than the default, you can set the sample period required for all the block outputs. This is useful when implementing features such as feedback loops in your design. In a feedback loop, it is not possible for Model Composer to determine a default sample rate, because the loop makes an input sample rate depend on a yet-to-be-determined output sample rate. Model Composer under these circumstances requires you to supply a hint to establish sample periods throughout a loop.

Use Behavioral HDL (otherwise use core)

When this checkbox is checked, the behavioral HDL generated by the M-code simulation is used instead of the structural HDL from the cores.

The M-code simulation creates the C simulation and this C simulation creates behavioral HDL. When this option is selected, it is this behavioral HDL that is used for further synthesis. When this option is not selected, the structural HDL generated from the cores and HDL templates (corresponding to each of the blocks in the model) is used instead for synthesis. Cores are generated for each block in a design once and cached for future netlisting. This capability ensures the fastest possible netlist generation while guaranteeing that the cores are available for downstream synthesis and place and route tools.

Use XtremeDSP Slice

This field specifies that if possible, use the XtremeDSP slice (DSP48 type element) in the target device. Otherwise, CLB logic are used for the multipliers.

Display shortened port names

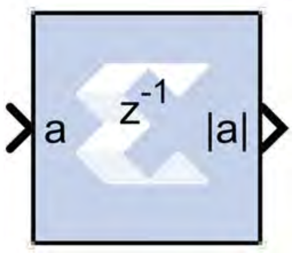
AXI4-Stream signal names have been shortened (by default) to improve readability on the block. Name shortening is purely cosmetic and when netlisting occurs, the AXI4-Stream name is used. For example, a shortened master signal on an AXI4-Stream interface might be data_tvalid. When you check **Display shortened port names**, the name becomes m_axis_data_tvalid.

Block Reference Pages

Following is an alphabetic listing of the blocks in the HDL blockset, with descriptions of each of the blocks.

Absolute

The Xilinx Absolute block outputs the absolute value of the input.



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**

- **Precision:** This parameter allows you to specify the output precision for fixed-point arithmetic. Floating point arithmetic output will always be **Full** precision.
    - **Full:** The block uses sufficient precision to represent the result without error.
    - **User Defined:** If you do not need full precision, this option allows you to specify a reduced number of total bits and/or fractional bits.
- **Fixed-point Output Type:**
    - **Arithmetic type:**
        - **Signed (2's comp):** The output is a Signed (2's complement) number.
        - **Unsigned:** The output is an Unsigned number.
- **Fixed-point Precision:**
    - **Number of bits:** Specifies the bit location of the binary point of the output number, where bit zero is the least significant bit.
    - **Binary point:** Position of the binary point. in the fixed-point output.
- **Quantization:**

  Refer to the section Overflow and Quantization.
- **Overflow:** Refer to the section Overflow and Quantization.

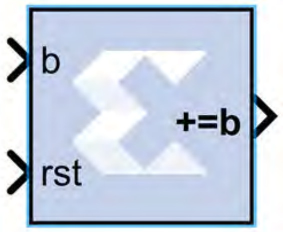
Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP Floating-Point Operator v7.1

Accumulator

The Xilinx® Accumulator block implements an adder or subtractor-based scaling accumulator.



The block's current input is accumulated with a scaled current stored value. The scale factor is a block parameter.

Block Interface

The block has an input `b` and an output `q`. The output must have the same width as the input data. The output will have the same arithmetic type and binary point position as the input. The output q is calculated as follows:

$$q(n)= \begin{cases} 0 & \text{if } rst=1 \\ q(n-1)xFeedbackScaling + b(n-1) & \text{otherwise} \end{cases}$$

A subtractor-based accumulator replaces addition of the current input `b(n)` with subtraction.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows:

  - **Operation:** This determines whether the block is adder- or subtractor-based.
  - **Fixed-Point Output Precision:**
    - **Number of bits:** specifies the bit location of the binary point of the output number, where bit zero is the least significant bit.
    - **Overflow:** Refer to the section Overflow and Quantization.
  - **Feedback scaling:** Specifies the feedback scale factor to be one of the following:

    1, 1/2, 1/4, 1/8, 1/16, 1/32, 1/64, 1/128, or 1/256.
  - **Optional Ports:**
    - **Provide synchronous reset port:** Activates an optional reset (rst) pin on the block. When the reset signal is asserted, the block goes back to its initial state. However, when a floating point accumulator is used, the output will be NAN during reset. The reset signal has precedence over the optional enable signal available on the block. The reset signal must run at a multiple of the block's sample rate. The signal driving the reset port must be Boolean.
  - **Bypass Option on Reset:**

        - **Reinitialize with input 'b':** When selected, the output of the accumulator is reset to the data on input port `b`. When not selected, the output of the accumulator is reset to zero. This option is available only when the block has a reset port. Using this option has clock speed implications if the accumulator is in a multirate system. In this case the accumulator is forced to run at the system rate because the clock enable (`CE`) signal driving the accumulator runs at the system rate, and the reset to input operation is a function of the `CE` signal.

- **Internal Precision tab:** Parameters specific to the Internal Precision tab are as follows:
    - **Floating Point Precision:**
        - **Input MSB Max:** The Most Significant Bit of the largest number that can be accepted.
        - **Output MSB Max:** The MSB of the largest result. It can be up to 54 bits greater than the Input MSB.
        - **Output LSB Min:** The Least Significant Bit of the smallest number that can be accepted. It is also the LSB of the accumulated result.
- **Implementation tab:** Parameters specific to the Implementation tab are as follows:
    - **Use behavioral HDL (otherwise use core):** The block is implemented using behavioral HDL. This gives the downstream logic synthesis tool maximum freedom to optimize for performance or area.
    - **Implement using:** Core logic can be implemented in **Fabric** or in a **DSP48**, if a DSP48 is available in the target device. The default is **Fabric**.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

The Accumulator block always has a latency of 1.

LogiCORE Documentation

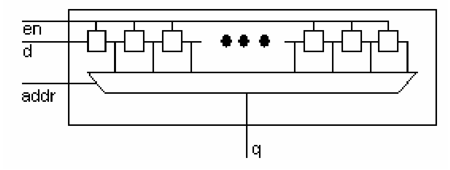
LogiCORE IP Accumulator v12.0

Addressable Shift Register

The Xilinx Addressable Shift Register block is a variable-length shift register in which any register in the delay chain can be addressed and driven onto the output data port.

The block operation is most easily thought of as a chain of registers, where each register output drives an input to a multiplexer, as shown below. The multiplexer select line is driven by the address port (addr). The output data port is shown below as q.

Figure 301: **Output Data Port**



The Addressable Shift Register has a maximum depth of 1024 and a minimum depth of 2. The address input port, therefore, can be between 1 and 10 bits (inclusive). The data input port width must be between 1 and 255 bits (inclusive) when this block is implemented with the Xilinx LogiCORE (for example, when **Use behavioral HDL (otherwise use core)** is unchecked).

In hardware, the address port is asynchronous relative to the output port. In the block S-function, the address port is therefore given priority over the input data port, for example, on each successive cycle, the addressed data value is read from the register and driven to the output before the shift operation occurs. This order is needed in the Simulink® software model to guarantee one clock cycle of latency between the data port and the first register of the delay chain. (If the shift operation were to come first, followed by the read, then there would be no delay, and the hardware would be incorrect.)

Block Interface

The block interface (inputs and outputs as seen on the Addressable Shift Register icon) are as follows:

d	data input
addr	address
en	enable signal (optional)

q	data output

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:** Parameters specific to this block are as follows:
  - **Infer maximum latency (depth) using address port width:** You can choose to allow the block to automatically determine the depth or maximum latency of the shift-register-based on the bit-width of the address port.
  - **Maximum latency (depth):** In the case that the maximum latency is not inferred (previous option), the maximum latency can be set explicitly.
  - **Initial value vector:** Specifies the initial register values. When the vector is longer than the shift register depth, the vector's trailing elements are discarded. When the shift register is deeper than the vector length, the shift register's trailing registers are initialized to zero.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

- **Implementation tab:** Parameters specific to this block are as follows:
  - **Optimization:** You can choose to optimize for **Resource** (minimum area) or for **Speed** (maximum performance).

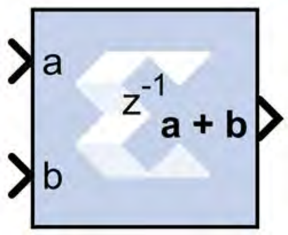
LogiCORE Documentation

LogiCORE IP RAM-based Shift Register v12.0

LogiCORE IP Floating-Point Operator v7.1

AddSub

The Xilinx AddSub block implements an adder/subtractor. The operation can be fixed (Addition or Subtraction) or changed dynamically under control of the sub mode signal.



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

- **Basic tab:** Parameters specific to the Basic tab are as follows:
    - **Operation:** Specifies the block operation to be Addition, Subtraction, or Addition/ Subtraction. When Addition/Subtraction is selected, the block operation is determined by the sub input port, which must be driven by a Boolean signal. When the sub input is 1, the block performs subtraction. Otherwise, it performs addition.
    - **Provide carry-in port:** When selected, allows access to the carry-in port, cin.
    - **Provide carry-out port:** When selected, allows access to the carry-out port, cout. The carry-out port is available only when **User defined** precision is selected, the inputs and output are unsigned, and the number of output integer bits equals x, where x = max (integer bits a, integer bits b).
    - **Latency:** The **Latency** value defines the number of sample periods by which the block's output is delayed. One sample period might correspond to multiple clock cycles in the corresponding FPGA implementation (for example, when the hardware is over-clocked with respect to the Simulink model). Model Composer will not perform extensive pipelining unless you select the **Pipeline for maximum performance** option (on the Implementation tab, described below); additional latency is usually implemented as a shift register on the output of the block.
- **Output tab:**
    - **Precision:**

      This parameter allows you to specify the output precision for fixed-point arithmetic. Floating point arithmetic output will always be **Full** precision.

        - **Full:** The block uses sufficient precision to represent the result without error.
        - **User Defined:** If you do not need full precision, this option allows you to specify a reduced number of total bits and/or fractional bits.
    - **Fixed-point Output Type:**
        - **Arithmetic Type:**
            - **Signed (2’s comp):** The output is a Signed (2’s complement) number.
            - **Unsigned:** The output is an Unsigned number.
        - **Fixed -point Precision:**
            - **Number of bits:** Specifies the bit location of the binary point of the output number, where bit zero is the least significant bit.

    - **Binary point:** Position of the binary point in the fixed-point output.
  - **Quantization:** Refer to the section Overflow and Quantization.
  - **Overflow:** Refer to the section Overflow and Quantization.
- **Implementation tab:**

  Parameters specific to the Implementation tab are as follows:

  - **Use behavioral HDL (otherwise use core):** The block is implemented using behavioral HDL. This gives the downstream logic synthesis tool maximum freedom to optimize for performance or area.

    Note: For Floating-point operations, the block always uses the Floating-point Operator core.
- **Core Parameters:**
  - **Pipeline for maximum performance:**

    The XILINX LogiCORE can be internally pipelined to optimize for speed instead of area. Selecting this option puts all user defined latency into the core until the maximum allowable latency is reached. If the **Pipeline for maximum performance** option is not selected and latency is greater than zero, a single output register is put in the core and additional latency is added on the output of the core.

    The **Pipeline for maximum performance** option adds the pipeline registers throughout the block, so that the latency is distributed, instead of adding it only at the end. This helps to meet tight timing constraints in the design.
  - **Implement using:** Core logic can be implemented in **Fabric**, or in a **DSP48**, if a DSP48 is available in the target device. The default is **Fabric**.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP Adder/Subtractor v12.0

LogiCORE IP Floating-Point Operator v7.1

Assert

The Xilinx Assert block is used to assert a rate and/or a type on a signal. This block has no cost in hardware and can be used to resolve rates and/or types in situations where designer intervention is required.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

Parameters specific to this block are as follows:

- **Type:**
  - **Assert type:** Specifies whether or not the block will assert that the type at its input is the same as the type specified. If the types are not the same, an error message is reported. This block is listed in the following Xilinx Blockset libraries: Floating-Point and Index.
  - **Specify type:** Specifies whether or not the type to assert is provided from a signal connected to an input port named type or whether it is specified **Explicitly** from parameters in the Assert block dialog box.
  - **Output Type:** Specifies the data type of the output. Can be **Boolean**, **Fixed-point**, or **Floating-point**.
  - **Arithmetic Type:** If the Output Type is specified as Fixed-point, you can select **Signed (2's comp)** or **Unsigned** as the Arithmetic Type.
    - **Fixed-point Precision:**
      - **Number of bits:** Specifies the bit location of the binary point of the output number, where bit zero is the least significant bit.
      - **Binary point:** Position of the binary point in the fixed-point output.
    - **Floating-point Precision:**
      - **Single:** Specifies single precision (32 bits).
      - **Double:** Specifies double precision (64 bits).
      - **Custom:** This block is listed in the following: Activates the field below so you can specify the Exponent width and the Fraction width.
      - **Exponent width:** Specify the exponent width.
      - **Fraction width:** Specify the fraction width.
- **Rate:**

- **Assert rate:** specifies whether or not the block will assert that the rate at its input is the same as the rate specified. If the rates are not the same, an error message is reported.
- **Specify rate:** Specifies whether or not the initial rate to assert is provided from a signal connected to an input port named `rate`, or whether it is specified **Explicitly** from the **Sample rate** parameter in the Assert block dialog box.

- **Provide output port:** Specifies whether or not the block will feature an output port. The type and/or rate of the signal presented on the output port is the type and/or rate specified for assertion.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

The **Output type** parameter in this block uses the same description as the Arithmetic Type described in the topic Common Options in Block Parameter Dialog Boxes.

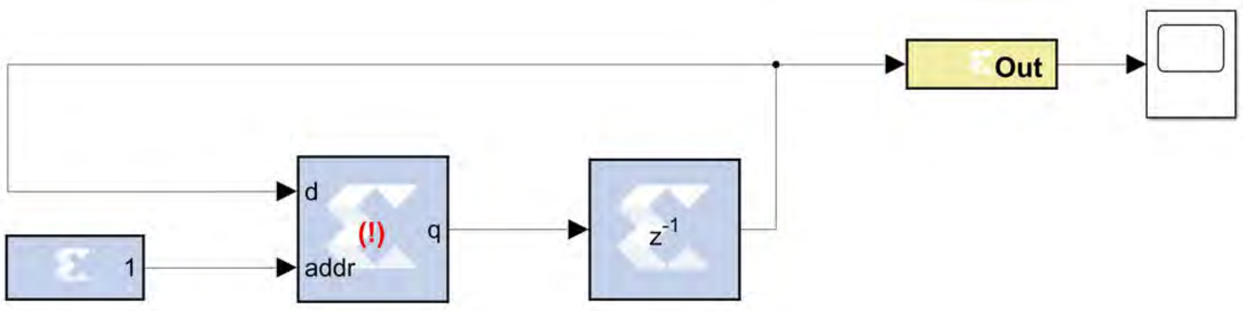
The Assert block does not use a Xilinx LogiCORE™ and does not use resources when implemented in hardware.

Using the Assert block to Resolve Rates and Types

In cases where the simulation engine cannot resolve rates or types, the Assert block can be used to force a particular type or rate. In general this might be necessary when using components that use feedback and act as a signal source. For example, the circuit below requires an Assert block to force the rate and type of an SRL16. In this case, you can use an Assert block to 'seed' the rate which is then propagated back to the SRL16 input through the SRL16 and back to the Assert block. The design below fails with the following message when the Assert block is not used.

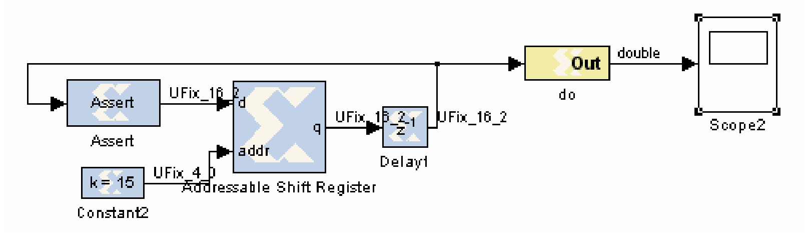
The data types could not be established for the feedback paths through this block. You might need to add Assert blocks to instruct the system how to resolve types.

Figure 302: **Addressable Shift Register**



To resolve this error, an Assert block is introduced in the feedback path as shown below:

Figure 303: **Addressable Shift Register with Assert Block**

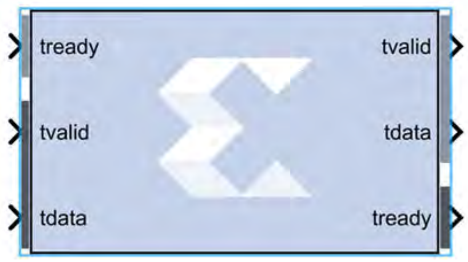


In the example, the Assert block is required to resolve the type, but the rate could have been determined by assigning a rate to the constant clock. The decision whether to use Constant blocks or Assert blocks to force rates is arbitrary and can be determined on a case by case basis.

Model Composer now resolves rates and types deterministically, however in some cases, the use of Assert blocks might be necessary for some Model Composer HDL components, even if they are resolvable. These blocks might include Black Box components and certain IP blocks.

AXI FIFO

The Xilinx AXI FIFO block implements a FIFO memory queue with an AXI-compatible block interface.



Block Interface

- **Write Channel:**
  - **tready:** Indicates that the slave can accept a transfer in the current cycle.
  - **tvalid:** Indicates that the master is driving a valid transfer. A transfer takes place when both tvalid and tready are asserted.
  - **tdata:** The primary input data channel.
- **Read Channel:**
  - **tdata:** The primary output for the data.
  - **tready:** Indicates that the slave can accept a transfer in the current cycle.

- **tvaild:** Indicates that the slave is accepting a valid transfer. A transfer takes place when both tvalid and tready are asserted.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows.

  - **Data Port Parameters:**
    - **FIFO depth:** Specifies the number of words that can be stored. Range 16-128K.
    - **Actual FIFO depth:** A report field that indicates the actual FIFO depth. The actual depth of the FIFO depends on its implementation and the features that influence its implementation.
  - **Optional Ports:**
    - **TDATA:** The primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes.
    - **TDEST:** Provides routing information for the data stream.
    - **TSTRB:** The byte qualifier that indicates whether the content of the associated byte of TDATA is processed as a data byte or a position byte. For a 64-bit DATA, bit 0 corresponds to the least significant byte on DATA, and bit 7 corresponds to the most significant byte. For example:
      - STROBE[0] = 1b, DATA[7:0] is valid
      - STROBE[7] = 0b, DATA[63:56] is not valid
    - **TREADY:** Indicates that the slave can accept a transfer in the current cycle.
    - **TID:** The data stream identifier that indicates different streams of data.
    - **TUSER:** The user-defined sideband information that can be transmitted alongside the data stream.
    - **TKEEP:** The byte qualifier that indicates whether the content of the associated byte of TDATA is processed as part of the data stream. Associated bytes that have the TKEEP byte qualifier de-asserted are null bytes and can be removed from the data stream. For a 64-bit DATA, bit 0 corresponds to the least significant byte on DATA, and bit 7 corresponds to the most significant byte. For example:
      - KEEP[0] = 1b, DATA[7:0] is a NULL byte
      - KEEP [7] = 0b, DATA[63:56] is not a NULL byte

- **TLAST:** Indicates the boundary of a packet.
- **arestn:** Adds arestn (global reset) port to the block.

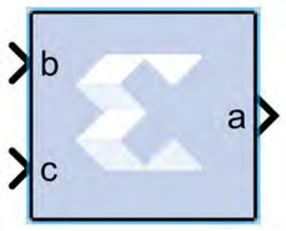
- **Data Threshold Parameters:**
  - **Provide FIFO occupancy DATA counts:** Adds data_count port to the block. This port indicates the number of words written into the FIFO. The count is guaranteed to never under-report the number of words in the FIFO, to ensure the user never overflows the FIFO. The exception to this behavior is when a write operation occurs at the rising edge of write clock; that write operation will only be reflected on WR_DATA_COUNT at the next rising clock edge. D = log2(FIFO depth)+1

- **Implementation tab:** FIFO Options
  - **FIFO implementation type:** Specifies how the FIFO is implemented in the FPGA. Possible options are: Common Clock block RAM and Common Clock Distributed RAM. The XPM_FIFO_AXIS macro will be inferred or implemented when the design is compiled. For information on the XPM_FIFO_AXIS Xilinx Parameterized Macro (XPM), refer to *UltraScale Architecture Libraries Guide* (UG974).

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

BitBasher

The Xilinx BitBasher block performs slicing, concatenation and augmentation of inputs attached to the block.



The operation to be performed is described using Verilog syntax which is detailed in this document. The block can have up to four output ports. The number of output ports is equal to the number of expressions. The block does not cost anything in hardware.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:** Parameters specific to the Basic tab are as follows.

- **BitBasher Expression:** Bitwise manipulation expression based on Verilog Syntax. Multiple expressions (limited to a maximum of 4) can be specified using new line as a separator between expressions.

- **Output Type tab:**
  - **Output:** This refers to the port on which the data type is specified.
  - **Output type:** Arithmetic type to be forced onto the corresponding output.
  - **Binary Point:** Binary point location to be forced onto the corresponding output.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Supported Verilog Constructs

The BitBasher block only supports a subset of Verilog expression constructs that perform bitwise manipulations including slice, concatenation, and repeat operators. All specified expressions must adhere to the following template expression:

```
output_var = {bitbasher_expr}
```

bitbasher_expr: A slice, concat or repeat expression based on Verilog syntax or simply an input port identifier.

output_var: The output port identifier. An output port with the name output_var will appear on the block and will hold the result of the wire expression bitbasher_expr.

Concat

```
output_var = {bitbasher_expr1, bitbasher_expr2, bitbasher_expr3}
```

The concat syntax is supported as shown above. Each of bitbasher_exprN could either be an expression or simply an input port identifier.

The following are some examples of this construct:

```
a1 = {b,c,d,e,f,g}
a2 = {e}
a3 = {b,{f,c,d},e}
```

Slice

```
output_var = {port_identifier[bound1:bound2]}(1)
output_var = {port_identifier[bitN]}(2)
```

port_identifier: The input port from which the bits are extracted.

bound1, bound2: Non-negative integers that lie between 0 and (bit-width of port_identifier – 1)

bitN: Non-negative integers that lie between 0 and (bit-width of port_identifier – 1)

As shown above, there are two schemes to extract bits from the input ports. If a range of consecutive bits need to be extracted, then the expression of the following form should be used.

output_var = {port_identifier[bound1:bound2]}(1)

If only one bit is to be extracted, then the alternative form should be used.

output_var = {port_identifier[bitN]}(2)

The following are some examples of this construct:

```
a1 = {b[7:3]}
```

a1 holds bits 7 through 3 of input b in the same order in which they appear in bit b (for example, if b is 110110110 then a1 is 10110).

```
a2 = {b[3:7]}
```

a2 holds bits 7 through 3 of input b in the reverse order in which they appear in bit b (for example, if b is 110100110 then a2 is 00101).

```
a3 = {b[5]}
```

a3 holds bit 5 of input b.

```
a4 = {b[7:5],c[3:9],{d,e}}
```

The above expression makes use of a combination of slice and concat constructs.Bits 7 through 5 of input b, bits 3 through 9 of input `c` and all the bits of `d` and `e` are concatenated.

Repeat

```
output_var = {N{bitbasher_expr}}
```

N: A positive integer that represents the repeat factor in the expression

The following are some examples of this construct:

```
a1 = {4{b[7:3]}}
```

The above expression is equivalent to a1 = {b[7:3], b[7:3], b[7:3], b[7:3]}

```
a2 = {b[7:3],2{c,d}}
```

The above expression is equivalent to a2 = {b[7:3],c,d,c,d}

Constants

Binary Constant: N'bbin_const

Octal Constant: N'ooctal_const

Decimal Constant: N'doctal_const

Hexadecimal Constant: N'hoctal_const

N: A positive integer that represents the number of bits that are used to represent the constant

bin_const: A legal binary number string made up of 0 and 1

octal_const: A legal octal number string made up of 0, 1, 2, 3, 4, 5, 6 and 7

decimal_const: A legal decimal number string made up of 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9

hexadecimal_const: A legal binary number string made up of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e and f

A constant can only be used to augment expressions already derived from input ports. In other words, a BitBasher block cannot be used to simply source constant like the Constant block.

The following examples make use of this construct:

```
a1 = {4'b1100, e}
```

if e were 110110110 then a1 would be 1100110110110.

```
a1 = {4'hb, e}
```

if e were 110110110 then a1 would be 1101110110110.

```
a1 = {4'o10, e}
```

if e were 110110110 then a1 would be 1000110110110.

Limitations

- Does not support masked parameterization on the bitbasher expressions.

- An expression cannot contain only constants, that is, each expression must include at least one input port.

Black Box

The Model Composer Black Box block provides a way to incorporate hardware description language (HDL) models into Model Composer.

The block is used to specify both the simulation behavior in Simulink and the implementation files to be used during code generation with Model Composer. A black box's ports produce and consume the same sorts of signals as other HDL blocks. When a black box is translated into hardware, the associated HDL entity is automatically incorporated and wired to other blocks in the resulting design.

The black box can be used to incorporate either VHDL or Verilog into a Simulink model. Black box HDL can be co-simulated with Simulink using the Model Composer interface to the Vivado® simulator.

In addition to incorporating HDL into a Model Composer model, the black box can be used to define the implementation associated with an external simulation model.

Requirements on HDL for Black Boxes

Every HDL component associated with a black box must adhere to the following Model Composer requirements and conventions:

- The entity name must not collide with any entity name that is reserved by Model Composer (e.g., xlfir, xlregister).
- Bi-directional ports are supported in HDL black boxes; however they will not be displayed in the Model Composer as ports, they will only appear in the generated HDL after netlisting.
- For a Verilog Black Box, the module and port names must be lower case, and follow standard Verilog naming conventions.
- For a VHDL Black Box, the supported port data types are std_logic and std_logic_vector.
- Top level ports should be ordered most significant bit down to least significant bit, as in std_logic_vector(7 downto 0), and not std_logic_vector(0 to 7).

- Top level ports with signed binary types in Verilog RTL are not supported (for example, `18'sb1010`). Only unsigned binary types are supported.
- Clock and clock enable ports must be named according to the conventions described below.
- Any port that is a clock or clock enable must be of type std_logic. (For Verilog black boxes, such ports must be non-vector inputs, e.g., input clk.)
- Clock and clock enable ports on a black box are not treated like other ports. When a black box is translated into hardware, Model Composer drives the clock and clock enable ports with signals whose rates can be specified according to the block's configuration and the sample rates that drive it in Simulink.
- Falling-edge triggered output data cannot be used.

IMPORTANT! *Model Composer does not import `.dcp` files as an IP for blackbox flows.*

To understand how clocks work for black boxes, it helps to understand how Model Composer handles Timing and Clocking. In general, to produce multiple rates in hardware, Model Composer uses a single clock along with multiple clock enables, one enable for each rate. The enables activate different portions of hardware at the appropriate times. Each clock enable rate is related to a corresponding sample period in Simulink. Every HDL block that requires a clock has at least one clock and clock enable port in its HDL counterpart. Blocks having multiple rates have additional clock and clock enable ports.

Clocks for black boxes work like those for other HDL blocks. The black box HDL must have a separate clock and clock enable port for each associated sample rate in Simulink. Clock and clock enable ports in black box HDL should be expressed as follows:

- Clock and clock enables must appear as pairs (for example, for every clock, there is a corresponding clock enable, and vice-versa). Although a black box can have more than one clock port, a single clock source is used to drive each clock port. Only the clock enable rates differ.
- Each clock name (respectively, clock enable name) must contain the substring clk (resp., ce).
- The name of a clock enable must be the same as that for the corresponding clock, but with ce substituted for clk. For example, if the clock is named src_clk_1, then the clock enable must be named src_ce_1.

Clock and clock enable ports are not visible on the black box block icon. A work around is required to make the top-level HDL clock enable port visible in Model Composer; the work around is to add a separate enable port to the top-level HDL and AND this signal with the actual clock enable signal.

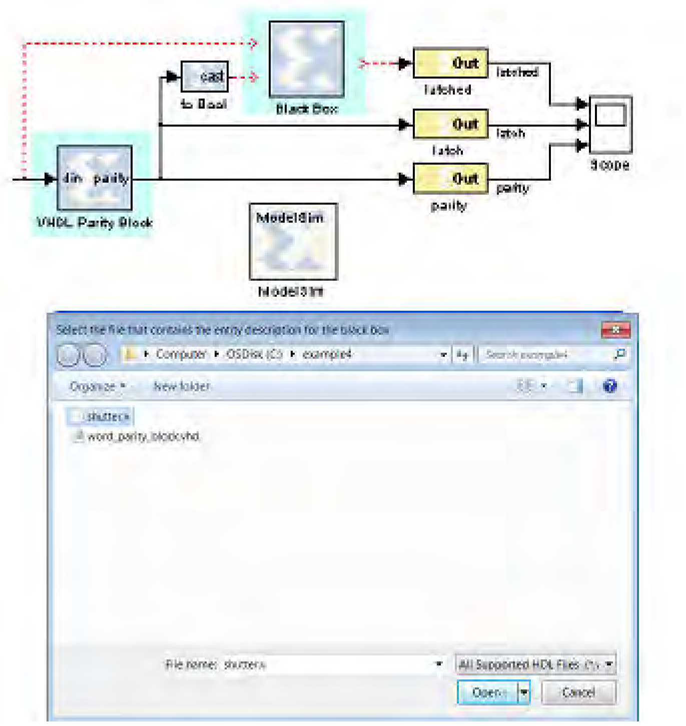
The Black Box Configuration Wizard

The Configuration Wizard is a tool that makes it easy to associate a Verilog or VHDL component to a black box. The wizard is invoked whenever a black box is added to a model.

IMPORTANT! *To use the wizard, copy the .v or .vhd file that defines the HDL component for a black box into the directory that contains the model.*

When a new black box is added to a model, the Configuration Wizard opens automatically. An example is shown in the figure below.

Figure 304: **Black Box Configuration Wizard Example**



From this wizard choose the HDL file that should be associated to the black box, then press the **Open** button. The wizard generates a configuration M-function (described below) for the black box, and associates the function with the block. The configuration M-function produced by the wizard can usually be used without change, but occasionally the function must be tailored by hand. Whether the configuration M-function needs to be modified depends on how complex the HDL is.

The Black Box Configuration M-Function

A black box must describe its interface (e.g., ports and generics) and its implementation to Model Composer. It does this through the definition of a MATLAB M-function (or p-function) called the block's configuration. The name of this function must be specified in the block parameter dialog box under the Block Configuration parameter.

The configuration M-function does the following:

- It specifies the top-level entity name of the HDL component that should be associated with the black box.

- It selects the language (for example, VHDL or Verilog).
- It describes ports, including type, direction, bit width, binary point position, name, and sample rate. Ports can be static or dynamic. Static ports do not change; dynamic ports change in response to changes in the design. For example, a dynamic port might vary its width and type to suit the signal that drives it.
- It defines any necessary port type and data rate checking.
- It defines any generics required by the black box HDL.
- It specifies the black box HDL and other files (e.g., EDIF) that are associated with the block.
- It defines the clocks and clock enables for the block (see the following topic on clock conventions).
- It declares whether the HDL has any combinational feed-through paths.

Model Composer provides an object-based interface for configuring black boxes consisting of two types of objects: BlockDescriptors, used to define entity characteristics, and PortDescriptors, used to define port characteristics. This interface is used to provide Model Composer information in the configuration M-function for black box about the block's interface, simulation model, and implementation.

If the HDL for a black box has at least one combinational path (for example, a direct feed-through from an input to an output port), the block must be tagged as combinational in its configuration M-function using the tagAsCombinational method. A black box can be a mixture (for example, some paths can be combinational while others are not).

IMPORTANT! *It is essential that a block containing a combinational path be tagged as such. Doing so allows Model Composer to identify such blocks to the Simulink simulator. If this is not done, simulation results are incorrect.*

The configuration M-function for a black box is invoked several times when a model is compiled. The function typically includes code that depends on the block's input ports. For example, sometimes it is necessary to set the data type and/or rate of an output port based on the attributes on an input port. It is sometimes also necessary to check the type and rate on an input port. At certain times when the function is invoked, Simulink might not yet know enough for such code to be executed.

To avoid the problems that arise when information is not yet known (in particular, exceptions), BlockDescriptor members *inputTypesKnown* and *inputRatesKnown* can be used. These are used to determine if Simulink is able, at the moment, to provide information about the input port types and rates respectively. The following code illustrates this point.

```
if (this_block.inputTypesKnown)
% set dynamic output port types
  % set generics that depend on input port types
  % check types of input ports
end
```

If all input rates are known, this code sets types for dynamic output ports, sets generics that depend on input port types, and verifies input port types are appropriate. Avoid the mistake of including code in these conditional blocks (e.g., a variable definition) that is needed by code outside of the conditional block.

Note that the code shown above uses an object named this_block. Every black box configuration M-function automatically makes this_block available through an input argument. In MATLAB, this_block is the object that represents the black box, and is used inside the configuration M-function to test and configure the black box. Every this_block object is an instance of the *SysgenBlockDescriptor* MATLAB class. The methods that can be applied to this_block are specified in Appendix A. A good way to generate example configuration M-function is to run the Configuration Wizard (described below) on simple VHDL entities.

Sample Periods

The output ports, clocks, and clock enables on a black box must be assigned sample periods in the configuration M-function. If these periods are dynamic, or the black box needs to check rates, then the function must obtain the input port sample periods. Sample periods in the black box are expressed as integer multiples of the system rate as specified by the *Simulink System Period* field on the System Generator token. For example, if the *Simulink System Period* is 1/8, and a black box input port runs at the system rate (for example, at 1/8), then the configuration M-function sees 1 reported as the port's rate. Likewise, if the *Simulink System Period* is specified as pi, and an output port should run four times as fast as the system rate (for example, at 4*pi), then the configuration M-function should set the rate on the output port to 4. The appropriate rate for constant ports is Inf.

As an example of how to set the output rate on each output port, consider the following code segment:

```
block.outport(1).setRate(theInputRate);
block.outport(2).setRate(theInputRate*5);
block.outport(3).setRate(theInputRate*5);
```

The first line sets the first output port to the same rate as the input port. The next two lines set the output rate to 5 times the rate of the input.

Block Parameters

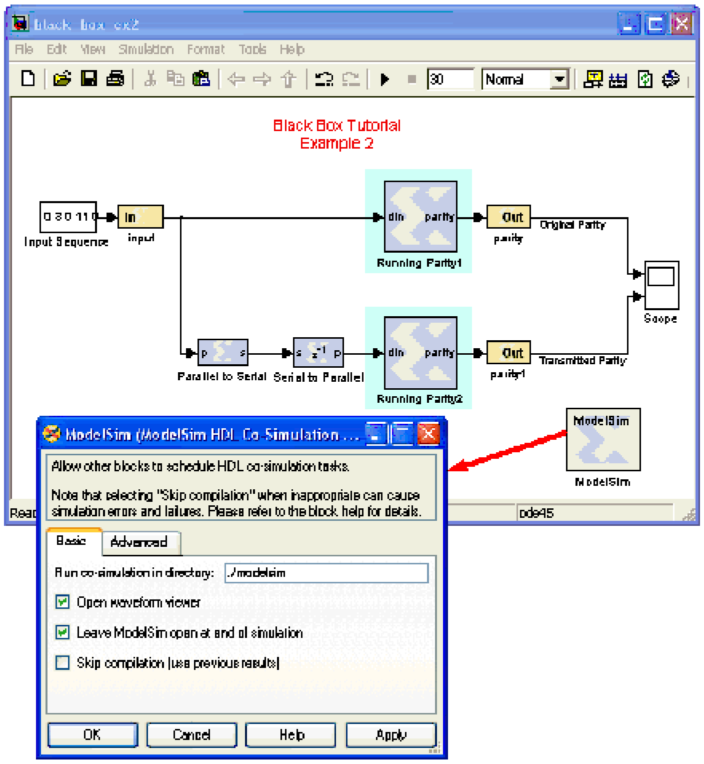
The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

- **Basic tab :** Parameters specific to the Basic tab are as follows.

- **Block Configuration M-Function:** Specifies the name of the configuration M-function that is associated to the black box. Ordinarily the file containing the function is stored in the directory containing the model, but it can be stored anywhere on the MATLAB path. Note that MATLAB limits all function names (including those for configuration M-functions) to 63 characters. Do not include the file extension (".m" or ".p") in the edit box.
- **Simulation Mode:** Tells the mode (Inactive, Vivado Simulator, or External co-simulator) to use for simulation. When the mode is Inactive, the black box ignores all input data and writes zeroes to its output ports. Usually for this mode the black box should be coupled, using a Configurable Subsystem.

Model Composer uses Configurable Subsystems to allow two paths to be identified – one for producing simulation results, and the other for producing hardware. This approach gives the best simulation speed, but requires that a simulation model be constructed. When the mode is Vivado Simulator or External co-simulator, simulation results for the black box are produced using co-simulation on the HDL associated with the black box. When the mode is External co-simulator, it is necessary to add a Questa HDL co-simulation block to the design, and to specify the name of the Questa block in the field labeled HDL Co-Simulator To Use. An example is shown below:

Figure 305: **Use of Configurable Subsystems Example**



Allow other blocks to schedule HDL co-simulation tasks.

Note that selecting "Skip compilation" when inappropriate can cause simulation errors and failures. Please refer to the block help for details.

Run co-simulation in directory: ./modelsim

☑ Open waveform viewer

☑ Leave ModelSim open at end of simulation

☐ Skip compilation (use previous results)

Model Composer supports the Questa simulator from Mentor Graphics®, Inc. for HDL co-simulation. For co-simulation of Verilog black boxes, a mixed mode license is required. This is necessary because the portion of the design that Model Composer writes is VHDL.

***Note*:** When you use the Questa simulator, the DefaultRadix used is Binary.

Usually the co-simulator block for a black box is stored in the same Subsystem that contains the black box, but it is possible to store the block elsewhere. The path to a co-simulation block can be absolute, or can be relative to the Subsystem containing the black box (e.g., "../Questa"). When simulating, each co-simulator block uses one license. To avoid running out of licenses, several black boxes can share the same co-simulation block. Model Composer automatically generates and uses the additional VHDL needed to allow multiple blocks to be combined into a single Questa simulation.

Data Type Translation for HDL Co-Simulation

During co-simulation, ports in Model Composer drive ports in the HDL simulator, and vice-versa. Types of signals in the tools are not identical, and must be translated. The rules used for translation are the following.

- A signal in Model Composer can be Boolean, unsigned or signed fixed point. Fixed-point signals can have indeterminate values, but Boolean signals cannot. If the signal's value is indeterminate in Model Composer, then all bits of the HDL signal become 'X', otherwise the bits become 0's and 1's that represent the signal's value.
- To bring HDL signals back into Model Composer, standard logic types are translated into Booleans and fixed-point values as instructed by the black box configuration M-function. When there is a width mismatch, an error is reported. Indeterminate signals of all varieties (weak high, weak low, etc.) are translated to Model Composer indeterminates. Any signal that is partially indeterminate in HDL simulation (e.g., a bit vector in which only the topmost bit is indeterminate) becomes entirely indeterminate in Model Composer.
- HDL to Model Composer translations can be tailored by adding a custom simulation-only top-level wrapper to the VHDL component. Such a wrapper might, for example, translate every weak low signal to 0 or every indeterminate signal to 0 or 1 before it is returned to Model Composer.

Example

The following is an example VHDL entity that can be associated to a HDL black box.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity word_parity_block is
  generic (width : integer := 8);
port (din : in std_logic_vector(width-1 downto 0);
  parity : out std_logic);
end word_parity_block;
architecture behavior of word_parity_block is
begin
  WORD_PARITY_Process : process (din)
  variable partial_parity : std_logic := '0';
  begin
  partial_parity := '0';
  XOR_BIT_LOOP: for N in din'range loop
```

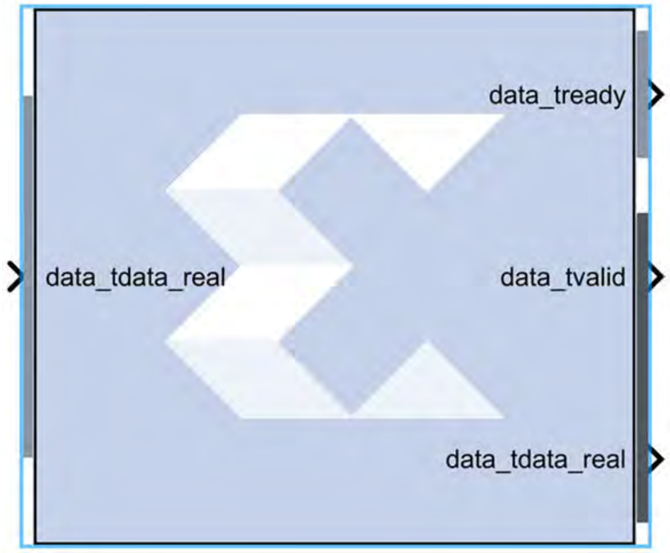
```
    partial_parity := partial_parity xor din(N);
    end loop; -- N
    parity <= partial_parity after 1 ns ;
    end process WORD_PARITY_Process;
end behavior;
```

The following is an example configuration M-function. It makes the VHDL shown above available inside a HDL black box.

```
function word_parity_block_config(this_block)
this_block.setTopLevelLanguage('VHDL');
    this_block.setEntityName('word_parity_block');
    this_block.tagAsCombinational;
    this_block.addSimulinkInport('din');
    this_block.addSimulinkOutport('parity');
    parity = this_block.port('parity');
    parity.setWidth(1);
    parity.useHDLVector(false);
    % ----------------------------
    if (this_block.inputTypesKnown)
    this_block.addGeneric('width',
    this_block.port('din').width);
    end  % if(inputTypesKnown)
    % ----------------------------
    % ----------------------------
    if (this_block.inputRatesKnown)
    din = this_block.port('din');
    parity.setRate(din.rate);
    end  % if(inputRatesKnown)
    % ----------------------------
    this_block.addFile('word_parity_block.vhd');
    return;
```

CIC Compiler 4.0

The Xilinx CIC Compiler provides the ability to design and implement AXI4-Stream-compliant Cascaded Integrator-Comb (CIC) filters for a variety of Xilinx FPGA devices.



CIC filters, also known as Hogenauer filters, are multi-rate filters often used for implementing large sample rate changes in digital systems. They are typically employed in applications that have a large excess sample rate. That is, the system sample rate is much larger than the bandwidth occupied by the processed signal as in digital down converters (DDCs) and digital up converters (DUCs). Implementations of CIC filters have structures that use only adders, subtractors, and delay elements. These structures make CIC filters appealing for their hardware-efficient implementations of multi-rate filtering.

Sample Rates and the CIC Compiler Block

The CIC Compiler block must always run at the system rate because the CIC Compiler block has a programmable rate change option and Simulink® cannot inherently support it. You should use the "ready" output signal to indicate to downstream blocks when a new sample is available at the output of the CIC Compiler block.

The CIC will downsample the data, but the sample rate will remain at the clock rate. If you look at the output of the CIC Compiler block, you will see each output data repeated R times for a rate change of R while the `data_tvalid` signal pulses once every R cycles. The downstream blocks can be clocked at lower-than-system rates without any problems as long as the clock is never slower than the rate change R.

There are several different ways this can be handled. You can leave the entire design running at the system rate then use registers with enables, or enables on other blocks to capture data at the correct time. Or alternatively, you can use a downsample block corresponding to the lowest rate change R, then again use enable signals to handle the cases when there are larger rate changes.

If there are not many required rate changes, you can use MUX blocks and use a different downsample block for each different rate change. This might be the case if the downstream blocks are different depending on the rate change, basically creating different paths for each rate. Using enables as described above will probably be the most efficient method.

If you are not using the CIC Compiler block in a programmable mode, you can place an up/down sample block after the CIC Compiler to correctly pass on the sample rate to downstream blocks that will inherit the rate and build the proper CE circuitry to automatically enable those downstream blocks at the new rate.

Block Parameters

- **Filter Specification tab:** Parameters specific to the Filter Specification tab are as follows.
    - **Filter Specification:**
        - **Filter Type:** The CIC core supports both interpolation and decimation architectures. When the filter type is selected as decimator the input sample stream is down-sampled by the factor *R*. When an interpolator is selected the input sample is up-sampled by *R*.
        - **Number of Stages:** Number of integrator and comb stages. If *N* stages are specified, there are *N* integrators and *N* comb stages in the filter. The valid range for this parameter is 3 to 6.
        - **Differential Delay:** Number of unit delays employed in each comb filter in the comb section of either a decimator or interpolator. The valid range of this parameter is 1 or 2.
        - **Number of Channels:** Number of channels to support in implementation. The valid range of this parameter is 1 to 16.
    - **Sample Rate Change Specification:**
        - **Sample Rate Changes:** Option to select between Fixed or Programmable.
        - **Fixed or Initial Rate(ir):** Specifies initial or fixed sample rate change value for the CIC. The valid range for this parameter is 4 to 8192.
        - **Minimum Rate:** The minimum rate change value for programmable rate change. The valid range for this parameter is 4 to fixed rate (ir).
        - **Maximum Rate:** The maximum rate change value for programmable rate change. The valid range for this parameter is fixed rate (ir) to 8192.
    - **Hardware Oversampling Specification:**
        - **Select format:** Choose Maximum_Possible, Sample_Period, or Hardware Oversampling Rate. Selects which method is used to specify the hardware oversampling rate. This value directly affects the level of parallelism of the block implementation and resources used. When "Maximum Possible" is selected, the block uses the maximum oversampling given the sample period of the signal connected to the Data field of the s_axis_data_tdata port. When you select "Hardware Oversampling Rate", you can specify the oversampling rate. When "Sample Period" is selected, the block clock is connected to the system clock and the value specified for the Sample Period parameter sets the input sample rate the block supports. The Sample Period parameter also determines the hardware oversampling rate of the block. When "Sample Period" is selected, the block is forced to use the s_axis_data_tvalid control port.

        - **Sample period:** Integer number of clock cycles between input samples. When the multiple channels have been specified, this value should be the integer number of clock cycles between the time division multiplexed input sample data stream.
        - **Hardware Oversampling Rate:** Enter the hardware oversampling rate if you select **Hardware_Oversampling_Rate** as the format.
- **Implementation tab:**
    - **Numerical Precision:**
        - **Quantization:** Can be specified as Full_Precision or Truncation.

          Note: Truncation occurs at the output stage only.
        - **Output Data Width:** Can be specified up to 48 bits for the Truncation option above.
    - **Optional:**
        - **Use Xtreme DSP slice:** This field specifies that if possible, use the XtremeDSP slice (DSP48 type element) in the target device.
        - **Use Streaming Interface:** Specifies whether or not to use a streaming interface for multiple channel implementations.
    - **Control Options:**
        - **ACLKEN:** Specifies if the block has a clock enable port (the equivalent of selecting the Has ACLKEN option in the CORE Generator GUI).
        - **ARESERTn:** Specifies that the block has a reset port. Active-Low synchronous clear. A minimum ARESETn pulse of two cycles is required.
        - **Has TREADY:** Specifies if the block has a TREADY port for the Data Output Channel (the equivalent of selecting the Has_DOUT_TREADY option in the CORE Generator GUI).

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP CIC Compiler 4.0

Clock Enable Probe

The Xilinx Clock Enable (CE) Probe provides a mechanism for extracting derived clock enable signals from Xilinx signals in Model Composer models.

The probe accepts any Xilinx signal type as input, and produces a Bool output signal. The Bool output can be used at any point in the design where Bools are acceptable. The probe output is a cyclical pulse that mimics the behavior of an ideal clock enable signal used in the hardware implementation of a multirate circuit. The frequency of the pulse is derived from the input signal's sample period. The enable pulse is asserted at the end of the input signal's sample period for the duration of one Simulink® system period. For signals with a sample period equal to the Simulink system period, the block's output is always one.

Shown below is an example model with an attached analysis scope that demonstrates the usage and behavior of the Clock Enable Probe. The Simulink system sample period for the model is specified in the System Generator token as 1.0 seconds. In addition to the Simulink system period, the model has three other sample periods defined by the Down Sample blocks. Clock Enable Probes are placed after each Down Sample block and extract the derived clock enable signal. The probe outputs are run to output gateways and then to the scope for analysis. Also included in the model is CLK probe that produces a Double representation of the hardware system clock. The scope output shows the output from the four Clock Enable probes in addition to the CLK probe output.

Figure 306: **Example Model with Attached Analysis Scope**

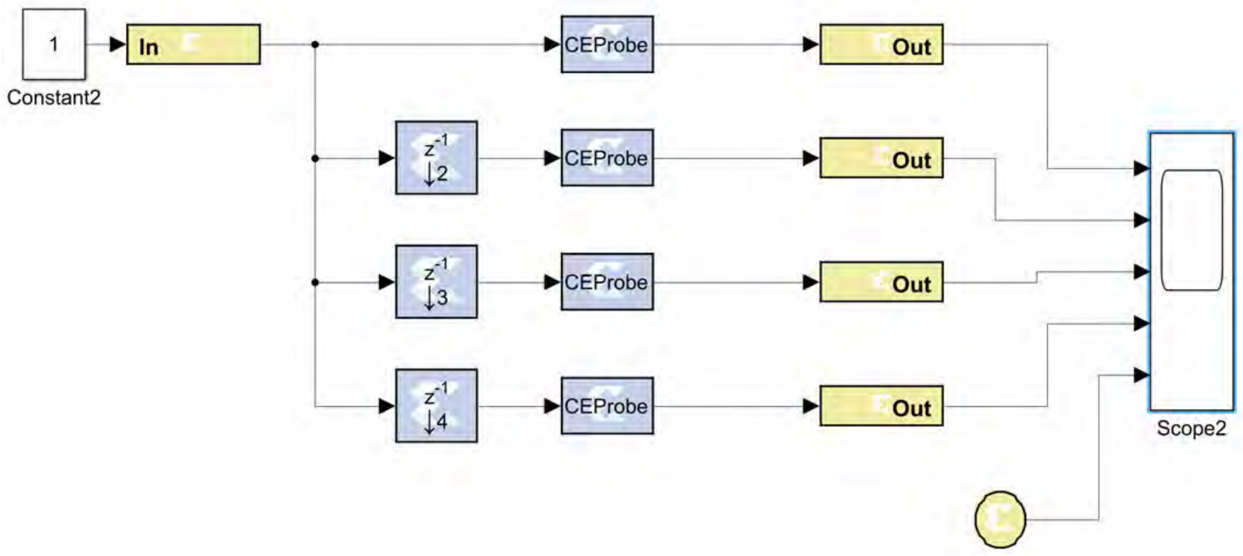
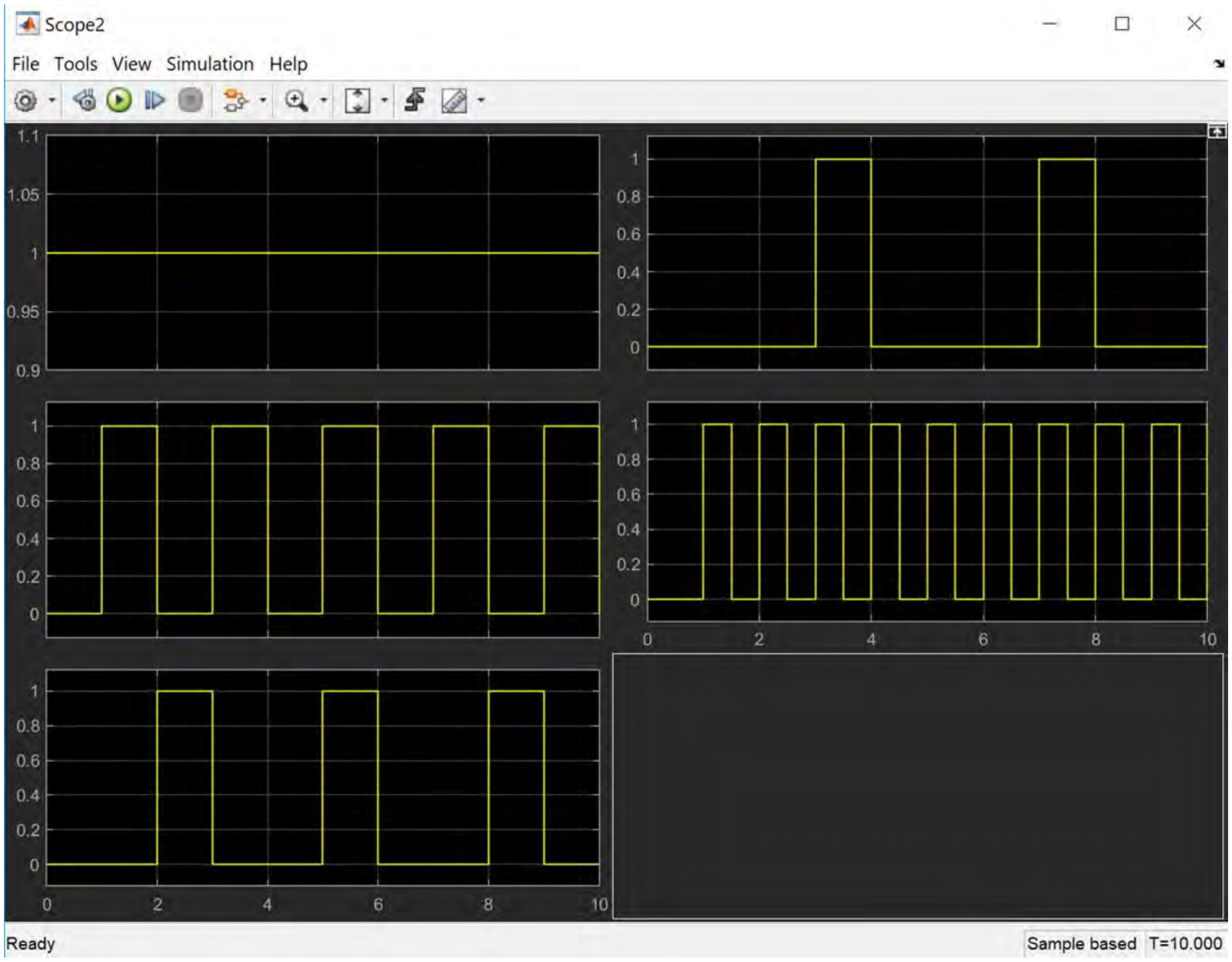


Figure 307: **Analysis Scope Output**



Options

- **Use clock enable signal without Multi-Cycle path constraints:** Used to disable multi-cycle path constraints on the generated signal from the Clock Enable Probe block. This is typically applied when the signal bring generated is used as separate timing signal that is not clock-enable related.

Clock Probe

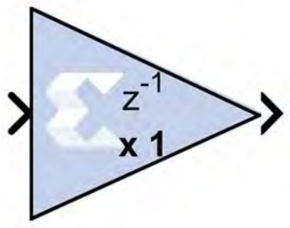
The Xilinx Clock Probe generates a double-precision representation of a clock signal with a period equal to the Simulink® system period.

The output clock signal has a 50/50 duty cycle with the clock asserted at the start of the Simulink sample period. The Clock Probe's double output is useful only for analysis, and cannot be translated into hardware.

There are no parameters for this block.

CMult

The Xilinx CMult block implements a *gain* operator, with output equal to the product of its input by a constant value. This value can be a MATLAB® expression that evaluates to a constant.



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows:

  - **Constant Type:**
    - **Fixed-point:** Use fixed-point data type.
    - **Floating-point:** Use floating-point data type. Can be a constant or an expression. If the constant cannot be expressed exactly in the specified fixed-point type, its value is rounded and saturated as needed.
  - **Fixed-point Precision:**
    - **Number of bits:** Specifies the bit location of the binary point of the constant, where bit zero is the least significant bit.
    - **Binary point:** Position of the binary point.

- **Floating-point Precision:**
    - **Single:** Specifies single precision (32 bits)
    - **Double:** Specifies double precision (64 bits)
    - **Custom:** Activates the field below so you can specify the Exponent width and the Fraction width.
    - **Exponent width:** Specify the exponent width.
    - **Fraction width:** Specify the fraction width.
- **Output tab:**
    - **Precision:** This parameter allows you to specify the output precision for fixed-point arithmetic. Floating point arithmetic output will always be Full precision.
        - **Full:** The block uses sufficient precision to represent the result without error.
        - **User Defined:** If you do not need full precision, this option allows you to specify a reduced number of total bits and/or fractional bits.
    - **Fixed-point Output Type:**
        - **Arithmetic type:**
            - **Signed (2's comp):** The output is a Signed (2's complement) number.
            - **Unsigned:** The output is an Unsigned number.
        - **Fixed-point Precision:**
            - **Number of bits:** Specifies the bit location of the binary point of the output number, where bit zero is the least significant bit.
            - **Binary point:** Position of the binary point in the fixed-point output.
        - **Quantization:**

          Refer to the section Overflow and Quantization.
        - **Overflow:**

          Refer to the section Overflow and Quantization.
- **Implementation tab:**

  Parameters specific to the Implementation tab are as follows.

- **Use behavioral HDL description (otherwise use core):** When selected, Model Composer uses behavioral HDL, otherwise it uses the Xilinx LogiCORE™ Multiplier. When this option is not selected (false) Model Composer internally uses the behavioral HDL model for simulation if any of the following conditions are true:
  - The constant value is 0 (or is truncated to 0).
  - The constant value is less than 0 and its bit width is 1.
  - The bit width of the constant or the input is less than 1 or is greater than 64.
  - The bit width of the input data is 1 and its data type is xlFix.
- **Core Parameters:**
  - **Implement using:** Specifies whether to use distributed RAM or block RAM.
  - **Test for optimum pipelining:** Checks if the Latency provided is at least equal to the optimum pipeline length supported for the given configuration of the block. Latency values that pass this test imply that the core produced is optimized for speed.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

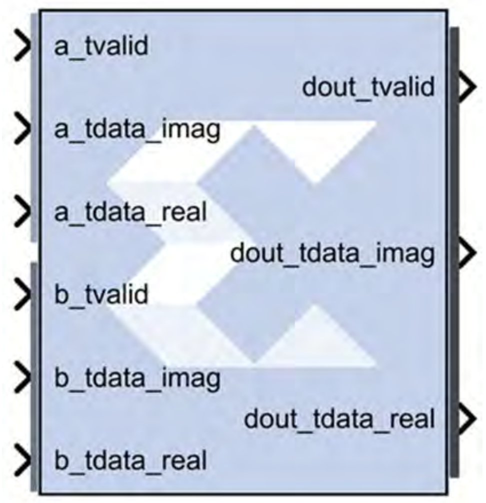
LogiCORE Documentation

LogiCORE IP Multiplier v12.0

LogiCORE IP Floating-Point Operator v7.1

Complex Multiplier 6.0

The Complex Multiplier block implements AXI4-Stream compliant, high-performance, optimized complex multipliers for devices based on user-specified options.



The two multiplicand inputs and optional rounding bit are input on independent AXI4-Stream channels as slave interfaces and the resulting product output using an AXI4-Stream master interface.

Within each channel, operands, and the results are represented in signed two's complement format. The operand widths. and the result width are parameterizable.

Block Parameters

- **Page 1 tab:**

  Parameters specific to the Basic tab are:

  - **Channel A Options:**
    - **Has TLAST:** Adds a tlast input port to the A channel of the block.
    - **Has TUSER:** Adds a tuser input port to the A channel of the block.
    - **TUSER Width:** User defined, maximum Limit range (1, 256).
  - **Channel B Options:**
    - **Has TLAST:** Adds a tlast input port to the B channel of the block.
    - **Has TUSER:** Adds a tuser input port to the B channel of the block.
    - **TUSER Width:** User defined. maximum Limit range (1, 256).
  - **Multiplier Construction Options:**
    - **Use_Mults:** Use embedded multipliers/XtremeDSP slices.
    - **Use_LUTs:** Use LUTs in the fabric to construct multipliers.

- **Optimization Goal:** Only available if Use_Mults is selected.
    - **Resources:** Uses the 3-real-multiplier structure. However, a 4-real-multiplier structure is used when the 3- l- multiplier structure uses more multiplier resources.
    - **Performance:** Always uses the 4-real multiplier structure to allow the best frequency performance to be achieved.
- **Flow Control Options:**
    - **Blocking:** Selects "Blocking" mode. In this mode, the lack of data on one input channel does block the execution of an operation if data is received on another input channel.
    - **NonBlocking:** Selects "Non-Blocking" mode. In this mode, the lack of data on one input channel does not block the execution of an operation if data is received on another input channel.

- **Page 2 tab:**
    - **Output Product Range:**

      Select the output bit width. The values are automatically set to provide the full-precision product when the A and B operand widths are set. The output is sign-extended if required.

      The natural output width for complex multiplication is (APortWidth + BPortWidth + 1). When the Output Width is set to be less than this, the most significant bits of the result are those output; the remaining bits will either be truncated or rounded according to Output Rounding option selected. That is to say, the output MSB is now fixed at (APortWidth + BPortWidth). For details please refer to the document LogiCORE IP Complex Multiplier v6.0 Product Guide.
    - **Output Rounding:** If rounding is required, the Output LSB must be greater than zero.
        - **Truncate:** Truncate the output.
        - **Random_Rounding:** When this option is selected, a ctrl_tvalid and ctrl_tdata input port is added to the block. Bit 0 if ctrl_tdata input determines the particular type if rounding for the operation. For details, refer to the Rounding section of the document LogiCORE IP Complex Multiplier v6.0 Product Guide.
    - **Channel CTRL Options:** The following options are activated when Random Rounding is selected.
        - **Has TLAST:** Adds a ctrl_tlast input port to the block.
        - **Has TUSER:** Adds a ctrl_user input port to the block.
        - **TUSER Width:** Specifies the bit width of the ctrl_tuser input port.
    - **Output TLAST Behavior:** Determines the behavior of the dout_tlast output port.
        - **Null:** Output is null.

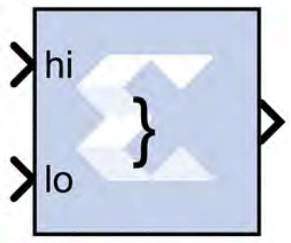
        - **Pass_A_TLAST:** Pass the value of the a_tlast input port to the dout_tlast output port.
        - **Pass B_TLAST:** Pass the value of the b_tlast input port to the dout_tlast output port.
        - **Pass CTRL_TLAST:** Pass the value of the ctrl_tlast input port to the dout_tlast output port.
        - **OR_all_TLASTS:** Pass the logical OR of all the present TLAST input ports.
        - **AND_all_TLASTS:** Pass the logical AND of all the present TLAST input ports.
    - **Core Latency:**
        - **Latency Configuration:**
            - **Automatic:** Block latency is automatically determined by Model Composer by pipelining the underlying LogiCORE™ for maximum performance.
            - **Manual:** You can adjust the block latency specifying the minimum block latency.
        - **Minimum Latency:** Entry field for manually specifying the minimum block latency.
    - **Control Signals:**
        - **ACLKEN:** Enables the clock enable (`aclken`) pin on the core. All registers in the core are enabled by this control signal.
        - **ARESETn:** Active-Low synchronous clear input that always takes priority over ACLKEN. A minimum ARESETn active pulse of two cycles is required, since the signal is internally registered for performance. A pulse of one cycle resets the core, but the response to the pulse is not in the cycle immediately following.
- **Advanced tab:** Block Icon Display
    - **Display shortened port names:** On by default. For example, when unchecked, **dout_tvalid** becomes **m_axis_dout_tvalid**.

LogiCORE Documentation

LogiCORE IP Complex Multiplier v6.0

Concat

The Xilinx Concat block performs a concatenation of n bit vectors represented by unsigned integer numbers, for example, n unsigned numbers with binary points at position zero.



The Xilinx Reinterpret block provides capabilities that can extend the functionality of the Concat block.

Block Interface

The block has *n* input ports, where n is some value between 2 and 1024, inclusively, and one output port. The first and last input ports are labeled `hi` and `low`, respectively. Input ports between these two ports are not labeled. The input to the hi port will occupy the most significant bits of the output and the input to the lo port will occupy the least significant bits of the output.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

Parameters specific to this block are as follows:

- Number of Inputs: specifies number of inputs, between 2 and 1024, inclusively, to concatenate together.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

The Concat block does not use a Xilinx LogiCORE.

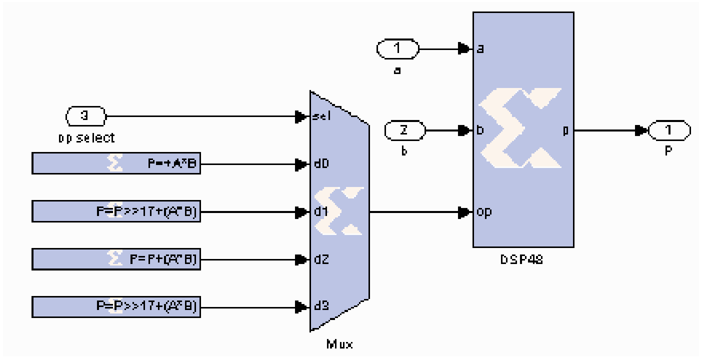
Constant

The Xilinx Constant block generates a constant that can be a fixed-point value, or a Boolean value. This block is similar to the Simulink® constant block, but can be used to directly drive the inputs on HDL blocks.

DSP48 Instruction Mode

The constant block, when set to create a DSP48 instruction, is useful for generating DSP48 control sequences. The following figure shows an example. The example implements a 35x35-bit multiplier using a sequence of four instructions in a DSP48 block. The constant blocks supply the desired instructions to a multiplexer that selects each instruction in the desired sequence.

Figure 308: **Example of Constant Block Creating a DSP48 Instruction**



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows:

- **Constant Value:**

  Specifies the value of the constant. When changed, the new value appears on the block icon. If the constant data type is specified as fixed-point and cannot be expressed exactly in the specified fixed-point type, its value is rounded and saturated as needed. A positive value is implemented as an unsigned number, a negative value as signed.

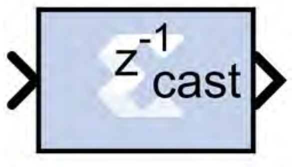
- **Output Type:** Specifies the data type of the output. Can be Boolean, Fixed-point, or Floating-point.
  - **Arithmetic Type:** If the Output Type is specified as Fixed-point, you can select **Signed (2's comp)**, **Unsigned**, or **DSP48 instruction** as the Arithmetic Type.
  - **Fixed-point Precision:**
    - **Number of bits:** Specifies the bit location of the binary point of the output number, where bit zero is the least significant bit.
    - **Binary point:** Position of the binary point in the fixed-point output.
  - **Floating-point Precision:**

        - **Single:** Specifies single precision (32 bits).
        - **Double:** Specifies double precision (64 bits)
        - **Custom:** Activates the field below so you can specify the Exponent width and the Fraction width.
        - **Exponent width:** Specifies the exponent width.
        - **Fraction width:** Specifies the fraction width.
    - **Sample Period:**
        - **Sampled Constant:** Allows a sample period to be associated with the constant output and inherited by blocks that the constant block drives. (This is useful mainly because the blocks eventually target hardware and the Simulink sample periods are used to establish hardware clock periods.)
- **DSP48 tab:**
    - **DSP48 Instruction:** The use of this block for DSP48 instructions is deprecated. Please use the Opmode block.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Convert

The Xilinx Convert block converts each input sample to a number of a desired arithmetic type. For example, a number can be converted to a signed (two's complement) or unsigned value.



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

- **Basic tab:** Parameters specific to the Basic Tab are as follows.
    - **Output Type:** Specify the output data type.
        - Boolean
        - Fixed-point

  - Floating-point

- **Arithmetic Type:** If the Output Type is specified as Fixed-point, you can select Signed (2's comp) or Unsigned.
- **Fixed-point Precision:**
  - **Number of bits:** Specifies the bit location of the binary point, where bit zero is the least significant bit.
  - **Binary point:** Specifies the bit location of the binary point.
- **Floating-point Precision:**
  - **Single:** Specifies single precision (32 bits).
  - **Double:** Specifies double precision (64 bits).
  - **Custom:** Activates the field below so you can specify the Exponent width and the Fraction width.
  - **Exponent width:** Specify the exponent width.
  - **Fraction width:** Specify the fraction width.
- **Quantization:**

  Quantization errors occur when the number of fractional bits is insufficient to represent the fractional portion of a value. The options are to **Truncate** (for example, to discard bits to the right of the least significant representable bit), or to **Round (unbiased: +/- inf)** or **Round (unbiased: even values)**.

  Round (unbiased: +/- inf) also known as "Symmetric Round (towards +/- inf)" or "Symmetric Round (away from zero)". This is similar to the MATLAB `round()` function. This method rounds the value to the nearest desired bit away from zero and when there is a value at the midpoint between two possible rounded values, the one with the larger magnitude is selected. For example, to round 01.0110 to a Fix_4_2, this yields 01.10, since 01.0110 is exactly between 01.01 and 01.10 and the latter is further from zero.

  Round (unbiased: even values) also known as "Convergent Round (toward even)" or "Unbiased Rounding". Symmetric rounding is biased because it rounds all ambiguous midpoints away from zero which means the average magnitude of the rounded results is larger than the average magnitude of the raw results. Convergent rounding removes this by alternating between a symmetric round toward zero and symmetric round away from zero. That is, midpoints are rounded toward the nearest even number. For example, to round 01.0110 to a Fix_4_2, this yields 01.10, since 01.0110 is exactly between 01.01 and 01.10 and the latter is even. To round 01.1010 to a Fix_4_2, this yields 01.10, since 01.1010 is exactly between 01.10 and 01.11 and the former is even.
- **Overflow:**

Overflow errors occur when a value lies outside the representable range. For overflow the options are to **Saturate** to the largest positive/smallest negative value, to **Wrap** (for example, to discard bits to the left of the most significant representable bit), or to **Flag as error** (an overflow as a Simulink error) during simulation. **Flag as error** is a simulation only feature. The hardware generated is the same as when **Wrap** is selected.

- **Optional Ports:**

  Provide enable port activates an optional enable (en) pin on the block. When the enable signal is not asserted the block holds its current state until the enable signal is asserted again or the reset signal is asserted.

- **Latency:**

  The Latency value defines the number of sample periods by which the block's output is delayed. One sample period might correspond to multiple clock cycles in the corresponding FPGA implementation (for example, when the hardware is over-clocked with respect to the Simulink® model). Model Composer will not perform extensive pipelining unless you select the **Pipeline for maximum performance** option (described below); additional latency is usually implemented as a shift register on the output of the block.

- **Implementation tab:** Parameters specific to the Implementation tab are as follows.

  - **Performance Parameters:**

    Pipeline for maximum performance: The XILINX LogiCORE can be internally pipelined to optimize for speed instead of area. Selecting this option puts all user defined latency into the core until the maximum allowable latency is reached. If the **Pipeline for maximum performance** option is *not* selected and latency is greater than zero, a single output register is put in the core and additional latency is added on the output of the core.

    The **Pipeline for maximum performance** option adds the pipeline registers throughout the block, so that the latency is distributed, instead of adding it only at the end. This helps to meet tight timing constraints in the design.

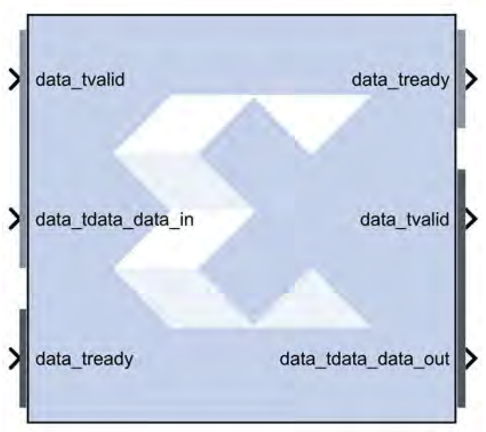
Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP Floating-Point Operator v7.1

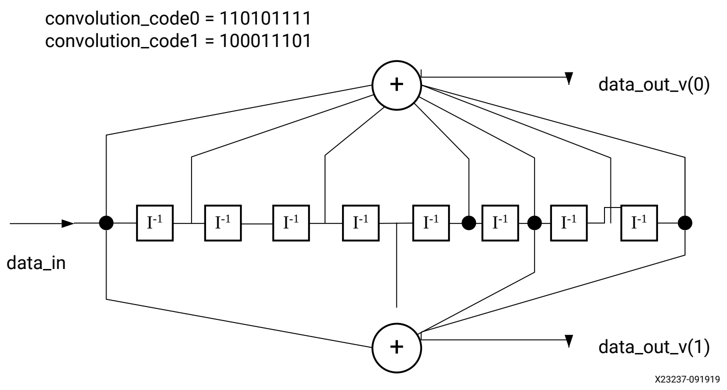
Convolution Encoder 9.0

The Xilinx® Convolution Encoder block implements an encoder for convolution codes. Ordinarily used in tandem with a Viterbi decoder, this block performs forward error correction (FEC) in digital communication systems. This block adheres to the AMBA® AXI4-Stream standard.



Values are encoded using a linear feed forward shift register which computes modulo-two sums over a sliding window of input data, as shown in the figure below. The length of the shift register is specified by the constraint length. The convolution codes specify which bits in the data window contribute to the modulo-two sum. Resetting the block will set the shift register to zero. The encoder rate is the ratio of input to output bit length; thus, for example a rate 1/2 encoder outputs two bits for each input bit. Similarly, a rate 1/ 3 encoder outputs three bits for each input bit.

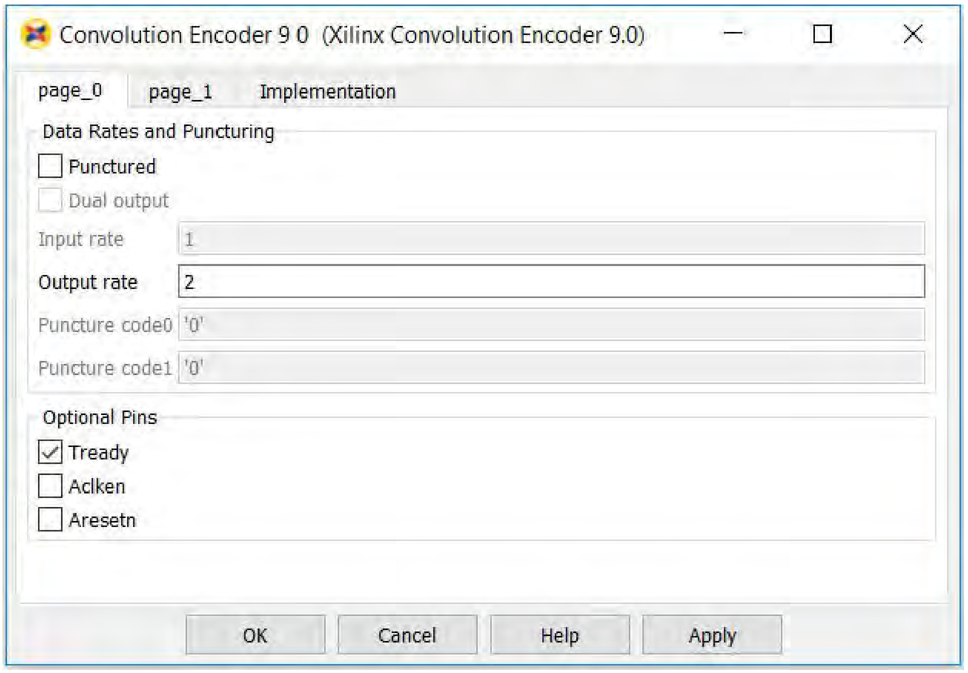
Figure 309: **Linear Feed Forward Shift Register**



Block Parameters

The following figure shows the block parameters dialog box.

Figure 310: **Block Parametere Dialog Box**


Convolution Encoder 9 0 (Xilinx Convolution Encoder 9.0)

page_0 | page_1 | Implementation

Data Rates and Puncturing

[ ] Punctured

[ ] Dual output

Input rate: 1

Output rate: 2

Puncture code0: '0'

Puncture code1: '0'

Optional Pins

[✓] Tready

[ ] Aclken

[ ] Aresetn

- **page_0 tab:**

  Parameters specific to the page_0 tab are as follows.

  - **Data Rates and Puncturing:**
    - **Punctured:** Determines whether the block is punctured.
    - **Dual Output:** Specifies a dual-channel punctured block.
    - **Input Rate:** Punctured: Only the input rate can be modified. Its value can range from 2 to 12, resulting in a rate n/m encoder where n is the input rate and n<m<2n.
    - **Output Rate:** Not Punctured: Only the output rate can be modified. Its value can be integer values from 2 to 7, resulting in a rate 1/2 or rate 1/7 encoder, respectively
    - **Puncture Code0 and Code1:** The two puncture pattern codes are used to remove bits from the encoded data prior to output. The length of each puncture code must be equal to the puncture input rate, and the total number of bits set to 1 in the two codes must equal the puncture output rate (m) for the codes to be valid. A 0 in any position indicates that the output bit from the encoder is not transmitted. See the associated LogiCORE™ data sheet for an example.
  - **Optional Pins:**
    - **Tready:** Adds a **tready** pin to the block. Indicates that the slave can accept a transfer in the current cycle.

- **Aclken:** Adds a aclken pin to the block. This signal carries the clock enable and must be of type Bool.
- **Aresetn:** Adds a **aresetn** pin to the block. This signal resets the block and must be of type `Bool`. The signal must be asserted for at least 2 clock cycles, however, it does not have to be asserted before the decoder can start decoding. If this pin is not selected, Model Composer ties this pin to inactive (high) on the core.

- **page_1 tab:** Parameters specific to the page_1 tab are as follows.
  - **Radix:**
    - **Convolution code radix:** Select Binary, Octal, or Decimal.
  - **Convolution:**
    - **Constraint length:** Constraint Length: Equals n+1, where n is the length of the constraint register in the encoder.
    - **Convolution code:** Array of binary convolution codes. Output rate is derived from the array length. Between 2 and 7 (inclusive) codes can be entered.

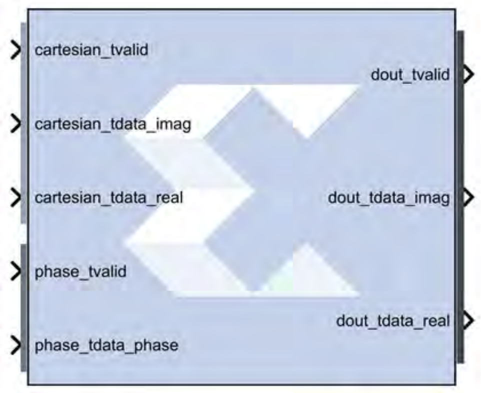
Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP Convolution Encoder 9.0

CORDIC 6.0

The Xilinx CORDIC block implements a generalized coordinate rotational digital computer (CORDIC) algorithm and is AXI compliant.



The CORDIC core implements the following equation types:

- Rotate
- Translate
- Sin_and_Cos
- Sinh_and_Cosh
- Arc_Tan
- Arc_Tanh
- Square_Root

Two architectural configurations are available for the CORDIC core:

- A word serial implementation with multiple-cycle throughput, but occupying a small silicon area.
- A fully parallel configuration with single-cycle data throughput at the expense of silicon area.

A coarse rotation is performed to rotate the input sample from the full circle into the first quadrant. (The coarse rotation stage is required as the CORDIC algorithm is only valid over the first quadrant). An inverse coarse rotation stage rotates the output sample into the correct quadrant.

The CORDIC algorithm introduces a scale factor to the amplitude of the result, and the CORDIC core provides the option of automatically compensating for the CORDIC scale factor.

Changes from CORDIC 4.0 to CORDIC 6.0

AXI compliant

- The CORDIC 6.0 block is AXI compliant.

Ports Renamed

- **en** to **aclken**
- **rst** to **aresetn**
- **rdy** maps to **dout_tready**. **cartesian_tready** and **phase_tready** are automatically added when their respective channels are added.
- **x_in** to **cartesian_tdata_real**
- **y_in** to **cartesian_tdata_imag**
- **phase_in** to **phase_tdata_phase**
- **x_out** to **dout_tdata_real**
- **y_out** to **dout_tdata_imag**
- **phase_out** to **dout_tdata_phase**

Port Changes

- The data output ports are not optional in CORDIC 6.0. The data output ports are selected based on the Function selected.
- A fully parallel configuration with single-cycle data throughput at the expense. There are separate **tuser**, **tlast**, and **tready** ports for the Cartesian and Phase input channels.
- The **dout_tlast** output port can be configured to provide **tlast** from the Cartesian input channel, from the Phase input channel, or the AND and or the OR of all **tlasts**.

Optimization

- When you select **Blocking** mode for the AXI behavior, you can then select whether the core is configured for minimum **Resources** or maximum **Performance**.

Displaying Port Names on the Block Icon

- You can select **Display shortened port names** to trim the length of the AXI port names on the block icon.

Block Parameters

- **Page 1 tab:**
    - **Functional selection:**
        - **Rotate:** When selected, the input vector, (real, imag), is rotated by the input angle using the CORDIC algorithm. This generates the scaled output vector, Zi * (real', imag').
        - **Translate:** When selected, the input vector (real, imag) is rotated using the CORDIC algorithm until the imag component is zero. This generates the scaled output magnitude, Zi * Mag(real, imag), and the output phase, Atan(imag/real).
        - **Sin_and_Cos:** When selected, the unit vector is rotated, using the CORDIC algorithm, by input angle. This generates the output vector (Cos( ), Sin( )).
        - **Sinh_and_Cosh:** When selected, the CORDIC algorithm is used to move the vector (1,0) through hyperbolic angle p along the hyperbolic curve. The hyperbolic angle represents the log of the area under the vector (real, imag) and is unrelated to a trigonometric angle. This generates the output vector (Cosh(p), Sinh(p)).
        - **Arc_Tan:** When selected, the input vector (real, imag) is rotated (using the CORDIC algorithm) until the imag component is zero. This generates the output angle, Atan(imag/real).
        - **Arc_Tanh:** When selected, the CORDIC algorithm is used to move the input vector (real, imag) along the hyperbolic curve until the imag component reaches zero. This generates the hyperbolic "angle," Atanh(imag/real). The hyperbolic angle represents the log of the area under the vector (real, imag) and is unrelated to a trigonometric angle.

- **Square_Root:** When selected a simplified CORDIC algorithm is used to calculate the positive square root of the input.

- **Architectural configuration:** Configuration:
  - **Word_Serial:** Select for a hardware result with a small area.
  - **Parallel:** Select for a hardware result with high throughput.
- **Pipelining mode:**
  - **No_Pipelining:** The CORDIC core is implemented without pipelining.
  - **Optimal:** The CORDIC core is implemented with as many stages of pipelining as possible without using any additional LUTs.
  - **Maximum:** The CORDIC core is implemented with a pipeline after every shift-add sub stage.
- **Data format:**
  - **SignedFraction:** Default setting. The real and imag inputs and outputs are expressed as fixed-point 2's complement numbers with an integer width of 2-bits.
  - **UnsignedFraction:** Available only for Square Root functional configuration. The real and imag inputs and outputs are expressed as unsigned fixed-point numbers with an integer width of 1-bit.
  - **UnsignedInteger:** Available only for Square Root functional configuration. The real and imag inputs and outputs are expressed as unsigned integers.
- **Phase format:**
  - **Radians:** The phase is expressed as a fixed-point 2's complement number with an integer width of 3-bits, in radian units.
  - **Scaled_Radians:** The phase is expressed as fixed-point 2's complement number with an integer width of 3-bits, with pi-radian units. One scaled-radian equals Pi * 1 radians.
- **Input/Output Options:**
  - **Input width:** Controls the width of the input ports **cartesian_tdata_real**, **cartesian_tdata_imag**, and **phase_tdata_phase**. The Input width range 8 to 48 bits.
  - **Output width:** Controls the width of the output ports **dout_tdata_real**, **dout_tdata_imag**, and **dout_tdata_phase**. The Output width range 8 to 48 bits.
- **Round mode:**
  - **Truncate:** The real, imag, and phase outputs are truncated.
  - **Round_Pos_Inf:** The real, imag, and phase outputs are rounded (1/2 rounded up).

- **Round_Pos_Neg_Inf:** The real, imag, and phase outputs are rounded (1/2 rounded up, -1/2 rounded down).
- **Nearest_Even:** The real, imag, and phase outputs are rounded toward the nearest even number (1/2 rounded down and 3/2 is rounded up).

- **Page 2 tab:**
  - **Advanced Configuration Parameters:**
    - **Iterations:** Controls the number of internal add-sub iterations to perform. When set to zero, the number of iterations performed is determined automatically based on the required accuracy of the output.
    - **Precision:** Configures the internal precision of the add-sub iterations. When set to zero, internal precision is determined automatically based on the required accuracy of the output and the number of internal iterations.
    - **Compensation scaling:** Controls the compensation scaling module used to compensate for CORDIC magnitude scaling. CORDIC magnitude scaling affects the Vector Rotation and Vector Translation functional configurations, and does not affect the SinCos, SinhCosh, ArcTan, ArcTanh and Square Root functional configurations. For the latter configurations, compensation scaling is set to No Scale Compensation.
    - **Coarse rotation:**

      Controls the instantiation of the coarse rotation module. Instantiation of the coarse rotation module is the default for the following functional configurations: Vector rotation, Vector translation, Sin and Cos, and Arc Tan. If Coarse Rotation is turned off for these functions then the input/output range is limited to the first quadrant (-Pi/4 to + Pi/4).

      Coarse rotation is not required for the Sinh and Cosh, Arctanh, and Square Root configurations. The standard CORDIC algorithm operates over the first quadrant. Coarse Rotation extends the CORDIC operational range to the full circle by rotating the input sample into the first quadrant and inverse rotating the output sample back into the appropriate quadrant.
  - **Optional ports:**
    - **Standard:**
      - **aclken:** When this signal is not asserted, the block holds its current state until the signal is asserted again or the aresetn signal is asserted. The aresetn signal has precedence over this clock enable signal. This signal has to run at a multiple of the blocks sample rate. The signal driving this port must be Boolean.

    - **aresetn:** When this signal is asserted, the block goes back to its initial state. This reset signal has precedence over the optional aclken signal available on the block. The reset signal has to run at a multiple of the block's sample rate. The signal driving this port must be Boolean.
    - **tready:** Adds **dout_tready** port if Blocking mode is activated.
  - **Cartesian:**
    - **tlast:** Adds a tlast input port to the Cartesian input channel.
    - **tuser:** Adds a tuser input port to the Cartesian input channel.
    - **tuser width:** Specifies the bit width of the Cartesian tuser input port.
  - **Phase:**
    - **tlast:** Adds a tlast input port to the Phase input channel.
    - **tuser:** Adds a tuser input port to the Phase input channel.
    - **tuser width:** Specifies the bit width of the Phase tuser input port.
  - **Tlast behavior:**
    - **Null:** Data output port.
    - **Pass_Cartesian_TLAST:** Data output port.
    - **Pass_Phase_TLAST:** Data output port.
    - **OR_all_TLASTS:** Pass the logical OR of all the present TLAST input ports.
    - **AND_all_TLASTS:** Pass the logical AND of all the present TLAST input ports
- **Flow control:**
  - **AXI behavior:**
    - **NonBlocking:** Selects "Non-Blocking" mode. In this mode, the lack of data on one input channel does not block the execution of an operation if data is received on another input channel.
    - **Blocking:** Selects "Blocking" mode. In this mode, the lack of data on one input channel does block the execution of an operation if data is received on another input channel.
  - **Optimization:** When NonBlocking mode is selected, the following optimization options are activated:
    - **Resources:** Core is configured for minimum resources.
    - **Performance:** Core is configured for maximum performance.

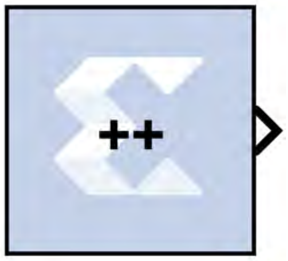
- **Implementation tab:**
  - **Block Icon Display:**
    - **Display shortened port names:** This option is ON by default. When unselected, the full AXI name of each port is displayed on the block icon.

LogiCORE Documentation

LogiCORE IP CORDIC v6.0

Counter

The Xilinx Counter block implements a free-running or count-limited type of an up, down, or up/down counter. The counter output can be specified as a signed or unsigned fixed-point number.



Free-running counters are the least expensive in FPGA hardware. The free-running up, down, or up/down counter can also be configured to load the output of the counter with a value on the input din port by selecting the **Provide Load Pin** option in the block's parameters.

$$out(n) = \begin{cases} InitialValue & if\ n=0 \\ (out(n-1)+Step)mod2^N & otherwise \end{cases}$$

The output for a free-running up counter is calculated as follows:

$$out(n) = \begin{cases} InitialValue & if\ n=0 \\ din(n-1) & if\ load(n-1)=1 \\ (out(n-1)+Step)mod2^N & otherwise \end{cases}$$

Here N denotes the number of bits in the counter. The free-running down counter calculations replace addition with subtraction.

For the free-running up/down counter, the counter performs addition when input up port is 1or subtraction when the input up port is 0.

A count-limited counter is implemented by combining a free-running counter with a comparator. Count limited counters are limited to only 64 bits of output precision. Count limited types of a counter can be configured to step between the initial and ending values, provided the step value evenly divides the difference between the initial and ending values.

The output for a count limited up counter is calculated as follows:

$$out(n) = \begin{cases} InitialValue & \text{if } n=0 \text{ or } out(n-1)=CountLimit \\ (out(n-1)+Step) mod 2^N & otherwise \end{cases}$$

The count-limited down counter calculation replaces addition with subtraction. For the count limited up/down counter, the counter performs addition when input up port is 1 or subtraction when input up port is 0.

The output for a free-running up counter with load capability is calculated as follows:

$$out(n) = \begin{cases} StartCount & \text{if } n=0 \text{ or } rst(n)=1 \\ din & \text{if } rst(n) = 0 \text{ and } load\ (n) = 1 \\ (out(n-1)+CountByValue) mod 2^N & otherwise \end{cases}$$

Here N denotes the number of bits in the counter. The down counter calculations replace addition by subtraction.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows:

  - **Counter type:** Specifies the counter to be a count-limited or free-running counter.
  - **Count to value:** Sspecifies the ending value, the number at which the count-limited counter resets. A value of `Inf` denotes the largest representable output in the specified precision. This cannot be the same as the initial value.
  - **Count direction:** Specifies the direction of the count (up or down) or provides an optional input port up (when up/down is selected) for specifying the direction of the counter.
  - **Initial value:** Specifies the initial value to be the output of the counter.
  - **Step:** Specifies the increment or decrement value.
  - **Output type:** Specifies the block output to be either Signed or Unsigned.
  - **Number of bits:** Specifies the number of bits in the block output.
  - **Binary point:** Specifies the location of the binary point in the block output.

  - **Provide load port:** When checked, the block operates as a free-running load counter with explicit load and din port. The load capability is available only for the free-running counter.
  - **Provide Synchronous reset port:** Activates an optional reset (rst) pin on the block. When the reset signal is asserted the block goes back to its initial state. Reset signal has precedence over the optional enable signal available on the block. The reset signal has to run at a multiple of the block's sample rate. The signal driving the reset port must be Boolean..

- **Implementation tab:**

  Parameters specific to the Implementation tab are as follows.

  - **Implementation Details:**
    - **Use behavioral HDL (otherwise use core):** The block is implemented using behavioral HDL. This gives the downstream logic synthesis tool maximum freedom to optimize for performance or area.Core Parameters
      - **Implement using:** Core logic can be implemented in **Fabric** or in a **DSP48**, if a DSP48 is available in the target device. The default is **Fabric**.

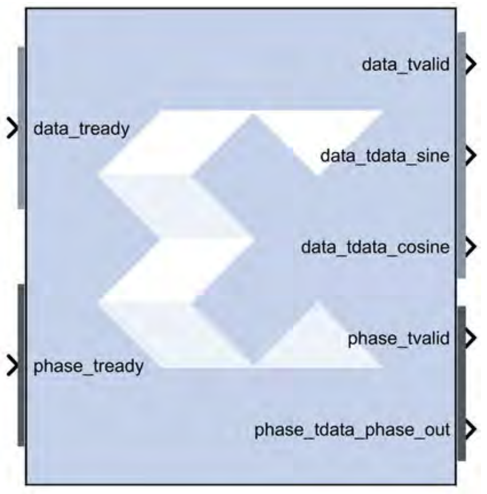
Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP Binary Counter v12.0

DDS Compiler 6.0

The Xilinx DDS (Direct Digital Synthesizer) Compiler block implements high performance, optimized Phase Generation and Phase to Sinusoid circuits with AXI4-Stream compliant interfaces for supported devices.

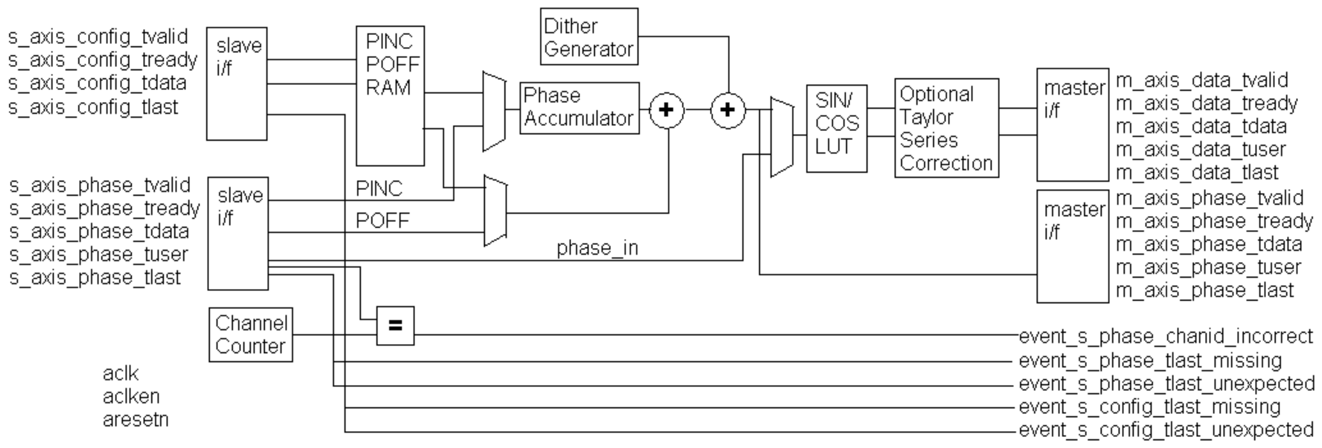


The core sources sinusoidal waveforms for use in many applications. A DDS consists of a Phase Generator and a SIN/COS Lookup Table (phase to sinusoid conversion). These parts are available individually or combined using this core.

Architecture Overview

To understand the DDS Compiler, it is necessary to know how the block is implemented in FPGA hardware. The following is a block diagram of the DDS Compiler core. The core consist of two main parts, a Phase Generator part and a SIN/COS LUT part. These parts can be used independently or together with an optional dither generator to create a DDS capability. A time-division multi-channel capability is supported with independently configurable phase increment and offset parameters.

Figure 311: **DDS Compiler Block Diagram**



Phase Generator

The Phase Generator consists of an accumulator followed by an optional adder to provide addition of phase offset. When the core is customized the phase increment and offset can be independently configured to be either fixed, programmable (using the CONFIG channel) or dynamic (using the input PHASE channel).

When set to fixed the DDS output frequency is set when the core is customized and cannot be adjusted once the core is embedded in a design.

When set to programmable, the CONFIG channel TDATA field will have a subfield for the input in question (PINC or POFF) or both if both have been selected to be programmable. If neither PINC nor POFF is set to programmable, there is no CONFIG channel.

When set to streaming, the input PHASE channel TDATA port (s_axis_phase_tdata) will have a subfield for the input in question (PINC or POFF) or both if both have been selected to be streaming. If neither PINC nor POFF is set to streaming, and the DDS is configured to have a Phase Generator then there is no input PHASE channel. Note that when the DDS is configured to be a SIN/COS Lookup only, the PHASE_IN field is input using the input PHASE channel TDATA port.

SIN/COS LUT

When configured as a SIN/COS Lookup only, the Phase Generator is not implemented, and the PHASE_IN signal is input using the input PHASE channel, and transformed into the SINE and COSINE outputs using a look-up table.

Efficient memory usage is achieved by exploiting the symmetry of sinusoid waveforms. The core can be configured for SINE only output, COSINE only output or both (quadrature) output. Each output can be configured independently to be negated. Precision can be increased using optional Taylor Series Correction. This exploits XtremeDSP slices on FPGA families that support them to achieve high SFDR with high speed operation.

AXI Ports that are Unique to this Block

Depending on the Configuration Options and Phase Increment/Offset Programmability options selected, different subfield-ports for the PHASE channel or the CONFIG channel (or both channels) are available on the block, as described in the table below.

Configuration Option	Phase Increment Programmability		Phase Offset Programmability	
	Option Selected	Available Port	Option Selected	Available Port
Phase_Generator_only Phase_Generator_and_SIN_COS_LUT	Programmable	s_axis_config_tdata_pinc	Programmable	s_axis_config_tdata_poff
	Streaming	s_axis_phase_tdata_pinc	Streaming	s_axis_phase_tdata_poff
	Fixed	NA	Fixed	NA
			None	NA
SIN_COS_LUT_only	In this configuration, input port s_axis_phase_tdata_phase_in are available			

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows.

  - **Configuration Options:** This parameter allows for two parts of the DDS to be instantiated separately or instantiated together. Select one of the following.
    - Phase_Generator_and_SIN_COS_LUT
    - Phase_Generator_only
    - SIN_COS_LUT_only
  - **System Requirements:**
    - **System Clock (MHz):** Specifies the frequency at which the block is clocked for the purposes of making architectural decisions and calculating phase increment from the specified output frequency. This is a fixed ratio off the System Clock.
    - **Number of Channels:** The channels are time-multiplexed in the DDS which affects the effective clock per channel. The DDS can support 1 to 16 time-multiplexed channels.
    - **Mode of Operation:**
      - **Standard:** The output frequency of the DDS waveform is a function of the system clock frequency, the phase width in the phase accumulator and the phase increment value.
      - **Rasterized:** The DDS does not truncate the accumulated phase. Rasterized operation is intended for configurations where the desired frequency is a rational fraction of the system clock (output frequency = system frequency * N/M, where 0 < N < M). Values of M from 9 to 16384 are supported.

    ***Note*:** Refer to the document LogiCORE IP DDS Compiler v6.0 Product Guide for a detailed explanation of these modes.
  - **Parameter Selection:** Select **System_Parameters** or **Hardware_Parameters**
  - **System Parameters:**
    - **Spurious Free Dynamic Range (dB):** The targeted purity of the tone produced by the DDS. This sets the output width as well as internal bus widths and various implementation decisions.
    - **Frequency Resolution (Hz):** This sets the precision of the PINC and POFF values. Very precise values will require larger accumulators. Less precise values will cost less in hardware resource.

- **Noise Shaping:** Select one: **None**, **Phase_Dithering**, **Taylor_Series_Corrected**, or **Auto**.

  If the Configuration Options selection is SIN_COS_LUT_only, then None and Taylor_Series_Corrected are the only valid options for Noise Shaping. If Phase_Generator_Only is selected, then None is the only valid choice for Noise Shaping.

- **Hardware Parameters:**
  - **Phase Width:** Equivalent to frequency resolution, this sets the width of the internal phase calculations.
  - **Output Width:** Broadly equivalent to SFDR, this sets the output precision and the minimum Phase Width allowable. However, the output accuracy is also affected by the choice of Noise Shaping.
- **Output Selection:**
  - **Sine_and_Cosine:** Place both a Sine and Cosine output port on the block.
  - **Sine:** Place only a Sine output port on the block.
  - **Cosine:** Place only a Cosine output port on the block.
    - **Polarity:**
      - **Negative Sine:** Negates the **sine** output.
      - **Negative Cosine:** Negates the **cosine** output.
- **Amplitude Mode:**
  - **Full_Range:** Selects the maximum possible amplitude.
  - **Unit_Circle:** Selects an exact power-of-two amplitude, which is about one half the Full_Range amplitude.

- **Implementation tab:**
  - **Implementation Options:**
    - **Memory Type:** Select between **Auto**, **Distributed_ROM**, or **Block_ROM**.
    - **Optimization Goal:** Select between **Auto**, **Area**, or **Speed**.
    - **DSP48 Use:** Select between **Minimal**, or **Maximal**. When set to Maximal, XtremeDSP slices are used to achieve to maximum performance.
  - **Latency Options:**
    - **Auto:** The DDS is fully pipelined for optimal performance.
    - **Configurable:** Allows you to select less pipeline stages in the **Latency** pulldown menu below. This generally results in less resources consumed.

- **Control Signals:**
    - **Has phase out:** When checked the DDS will have the `phase_output` port. This is an output of the Phase_Generator half of the DDS, so it precedes the sine and cosine outputs by the latency of the sine/cosine lookup table.
    - **ACLKEN:** Enables the clock enable (aclken) pin on the core. All registers in the core are enabled by this control signal.
    - **ARESETn:** Active-low synchronous clear input that always takes priority over ACLKEN. A minimum ARESETn active pulse of two cycles is required, since the signal is internally registered for performance. A pulse of one cycle resets the core, but the response to the pulse is not in the cycle immediately following.
- **Explicit Sample Period:**
    - **Use explicit period:** When checked, the DDS Compiler block uses the explicit sample period that is specified in the dialog entry box below.

- **AXI Channel Options tab:**
    - **AXI Channel Options:**
        - **TLAST:**

          Enabled when there is more than one DDS channel (as opposed to AXI channel), as TLAST is used to denote the transfer of the last time-division multiplied channel of the DDS. Options are as follows.

            - **Not_Required:** In this mode, no TLAST appears on the input PHASE channel nor on the output channels.
            - **Vector_Framing:** In this mode, TLAST on the input PHASE channel and output channels denotes the last.
            - **Packet_Framing:** In this mode, TLAST is conveyed from the input PHASE channel to the output channels with the same latency as TDATA. The DDS does not use or interpret the TLAST signal in this mode.This mode is intended as a service to ease system design for cases where signals must accompany the datastream, but which have no application in the DDS.
            - **Config_Triggered:** This is an enhanced variant of the Vector Framing option. In this option, the TLAST on the input PHASE channel can trigger the adoption of new configuration data from the CONFIG channel when there is new configuration data available. This allows the re-configuration to be synchronized with the cycle of time-division-multiplexed DDS channels.
        - **TREADY:**

        - **Output TREADY:** When selected, the output channels will have a TREADY and hence support the full AXI handshake protocol with inherent back-pressure. If there is an input PHASE channel, its TREADY is also determined by this control, so that the datapath from input PHASE channel to output channels as a whole supports backpressure or not.
    - **TUSER Options:** Select one of the following options for the **Input**, **DATA Output**, and **PHASE Output**.
        - **Not_Required:** Neither of the above uses is required; the channel in question will not have a TUSER field.
        - **Chan_ID_Field:** In this mode, the TUSER field identifies the time-division-multiplexed channel for the transfer.
        - **User_Field:** In this mode, the block ignores the content of the TUSER field, but passes the content untouched from the input PHASE channel to the output channels.
        - **User and Chan_ID_Field:** In this mode, the TUSER field has both a `user` field and a `chan_id` field, with the `chan_id` field in the least significant bits. The minimal number of bits required to describe the channel will determine the width of the `chan_id field`. For example, 7 channels will require 3 bits.
        - **User Field Width:** This field determines the width of the bit field which is conveyed from input to output untouched by the DDS.
    - **Config Channel Options:**
        - **Synchronization Mode:**
            - **On_Vector:** In this mode, the re-configuration data is applied when the channel starts a new cycle of time-division-multiplexed channels.
            - **On_Packet:** In this mode, available when TLAST is set to packet framing, the TLAST channel will trigger the re-configuration. This mode is targeted at the case where it is to be associated with the packets implied by the input TLAST indicator.
- **Output Frequency tab:**
    - **Phase Increment Programmability:**

      Specifies the phase increment to be **Fixed**, **Programmable** or **Streaming**. The choice of Programmable adds channel, data, and we input ports to the block.

      The following fields are activated when Phase_Generator_and_SIN_COS_LUT is selected as the Configuration Options field on the Basic tab, the Parameter Selection on the Basic tab is set to Hardware Parameters and Phase Increment Programmability field on the Phase Offset Angles tab is set to **Fixed** or **Programmable**.

        - **Output frequencies (MHz):** For each channel, an independent frequency can be entered into an array. This field is activated when Parameter Selection on the Basic tab is set to **System Parameters** and Phase Increment Programmability is **Fixed** or **Programmable**.
        - **Phase Angle Increment Values:** This field is activated when Phase_Generator_and_SIN_COS_LUT is selected as the Configuration Options field on the Basic tab, the Parameter Selection on the Basic tab is set to **Hardware Parameters** and Phase Increment Programmability field on the Phase Offset Angles tab is set to **Fixed** or **Programmable**. Values must be entered in binary. The range is 0 to the weight of the accumulator, for example, 2Phase_Width-1.

- **Phase Offset Angles tab:**
    - **Phase Offset Programmability:** Specifies the phase offset to be **None**, **Fixed**, **Programmable** or **Streaming**. The choice of Fixed or Programmable adds the channel, data, and we input ports to the block.
        - **Phase Offset Angles (x2pi radians):** For each channel, an independent offset can be entered into an array. The entered values are multiplied by 2π radians. This field is activated when Parameter Selection on the Basic tab is set to System Parameters and Phase Increment Programmability is **Fixed** or Programmable.
        - **Phase Angle Offset Values:** For each channel, an independent offset can be entered into an array. The entered values are multiplied by 2π radians. This field is activated when Parameter Selection on the Basic tab is set to Hardware Parameters and Phase Increment Programmability is Fixed or Programmable.
- **Advanced tab:** Block Icon Display
    - **Display shortened port names:** This option is ON by default. When unselected, the full AXI name of each port is displayed on the block.

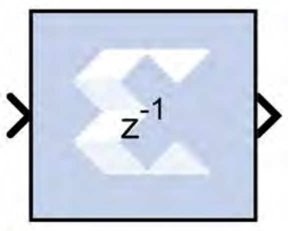
Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP DDS Compiler v6.0 Product Guide

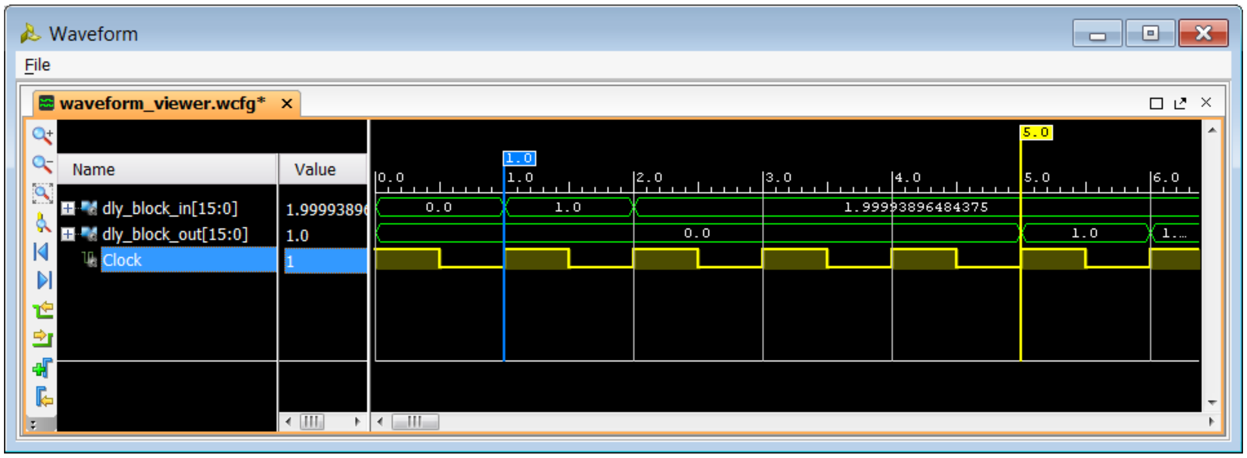
Delay

The Xilinx Delay block implements a fixed delay of L cycles.



The delay value is displayed on the block in the form z^{-L}, which is the *Z-transform* of the block's transfer function. Any data provided to the input of the block will appear at the output after L cycles. The rate and type of the data of the output is inherited from the input. This block is used mainly for matching pipeline delays in other portions of the circuit. The delay block differs from the register block in that the register allows a latency of only 1 cycle and contains an initial value parameter. The delay block supports a specified latency but no initial value other than zeros.The figure below shows the Delay block behavior when **L=4** and **Period=1s**.

Figure 312: **Delay Block Behavior**



For delays that need to be adjusted during run-time, you should use the **Addressable Shift Register** block. Delays that are not an integer number of clock cycles are not supported and such delays should not be used in synchronous design (with a few rare exceptions).

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows:

  - **Provide synchronous reset port:** this option activates an optional reset (rst) pin on the block. When the reset signal is asserted the block goes back to its initial state. Reset signal has precedence over the optional enable signal available on the block. The reset signal has to run at a multiple of the block's sample rate. The signal driving the reset port must be Boolean.
  - **Provide enable port::** this option activates an optional enable (en) pin on the block. When the enable signal is not asserted the block holds its current state until the enable signal is asserted again or the reset signal is asserted. Reset signal has precedence over the enable signal. The enable signal has to run at a multiple of the block 's sample rate. The signal driving the enable port must be Boolean.
- **Latency:** Latency is the number of cycles of delay. The latency can be zero, provided that the **Provide enable port** checkbox is not checked. The latency must be a non-negative integer. If the latency is zero, the delay block collapses to a wire during logic synthesis. If the latency is set to L=1, the block will generally be synthesized as a flip-flop (or multiple flip-flops if the data width is greater than 1).
- **Implementation tab:**

  Parameters specific to the Implementation tab are as follows:

  - **Implement using behavioral HDL:** Uses behavioral HDL as the implementation. This allows the downstream logic synthesis tool to choose the best implementation.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

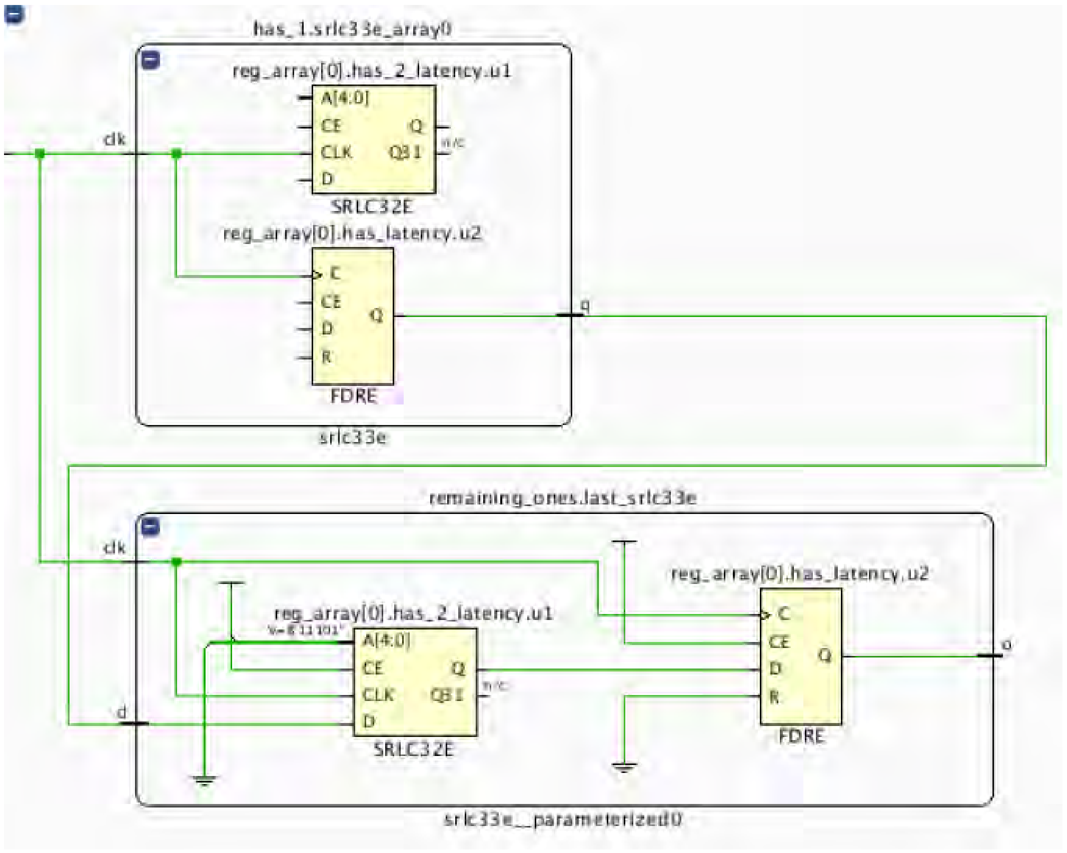
Logic Synthesis using Behavioral HDL

This setting is recommended if you are using Synplify Pro as the downstream logic synthesis tool. The logic synthesis tool will implement the delay as it desires, performing optimizations such as moving parts of the delay line back or forward into blockRAMs, DSP48s, or embedded IOB flip-flops; employing the dedicated SRL cascade outputs for long delay lines based on the architecture selected; and using flip-flops to terminate either or both ends of the delay line based on path delays. Using this setting also allows the logic synthesis tool, if sophisticated enough, to perform retiming by moving portions of the delay line back into combinational logic clouds.

Logic Synthesis using Structural HDL

If you do not check the box **Implement using behavioral HDL**, then structural HDL is used. This is the default setting and results in a known, but less-flexible, implementation which is often better for use with Vivado synthesis. In general, this setting produces structural HDL comprising an SRL (Shift-Register LUT) delay of (L-1) cycles followed by a flip-flop, with the SRL and the flip-flop getting packed into the same slice. For a latency greater than L=33, multiple SRL/flip-flop sets are cascaded, albeit without using the dedicated cascade routes. For example, the following is the synthesis result for a 1-bit wide delay block with a latency of L=64:

Figure 313: **1-Bit Wide Delay Block with a Latency of L=64**

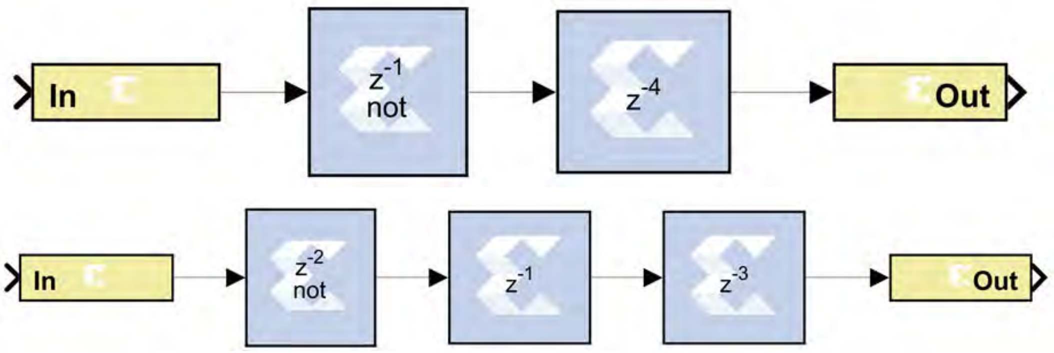

has_1.srlc33e_array0
reg_array[0].has_2_latency.u1
A[4:0]
CE
CLK
D
Q
Q31
n/c
SRLC32E
clk
reg_array[0].has_latency.u2
C
CE
D
R
Q
FDRE
q
srlc33e
remaining_ones.last_srlc33e
clk
reg_array[0].has_latency.u2
reg_array[0].has_2_latency.u1
SRLC32E
d
FDRE
o
srlc33e__parameterized0

The first SRL provides a delay of 32 cycles and the associated flip-flop adds another cycle of delay. The second SRL provides a delay of 30 cycles; this is evident because the address is set to {A4,A3,A2,A1,A0}=11101 (binary) = 29, and the latency through an SRL is the value of the address plus one. The last flip-flop adds a cycle of delay, making the grand total L=32+1+30+1=64 cycles.

The SRL is an efficient way of implementing delays in the Xilinx architecture. An SRL and its associated flip-flop that comprise a single *logic cell* can implement 33 cycles of delay whereas a delay line consisting only of flip-flops can implement only one cycle of delay per logic cell.

The SRL has a setup time that is longer than that of a flip-flop. Therefore, for very fast designs with a combinational path preceding the delay block, it can be advantageous, when using the structural HDL setting, to precede the delay block with an additional delay block with a latency of L=1. This ensures that the critical path is not burdened with the long setup time of the SRL. An example is shown below.

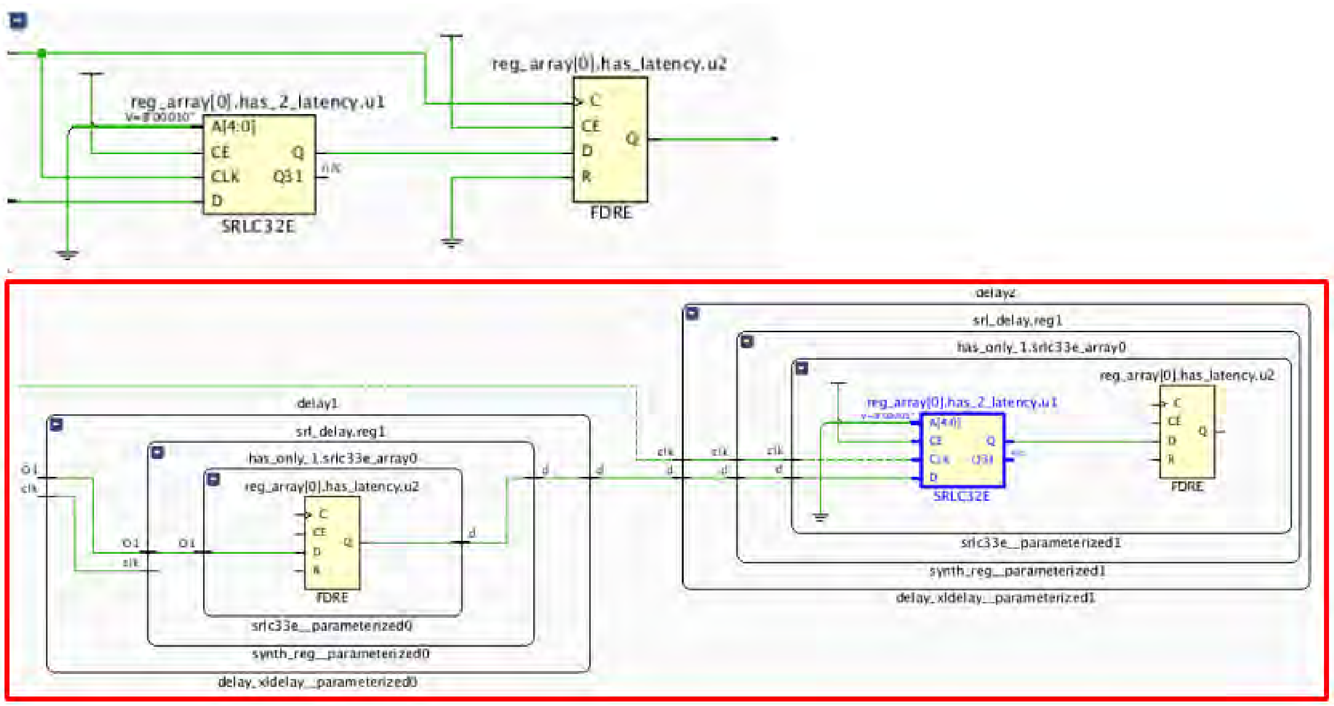
Figure 314: **Delay Block with an Additional Delay Block with a Latency**



In the example, the two designs are logically equivalent, but the bottom one will have a faster hardware implementation. The bottom design will have the combinational path formed by Inverter1 terminated by a flip-flop, which has a shorter setup time than an SRL.

The synthesis results of both designs are shown below, with the faster design highlighted in red:

Figure 315: **Synthesis Results**

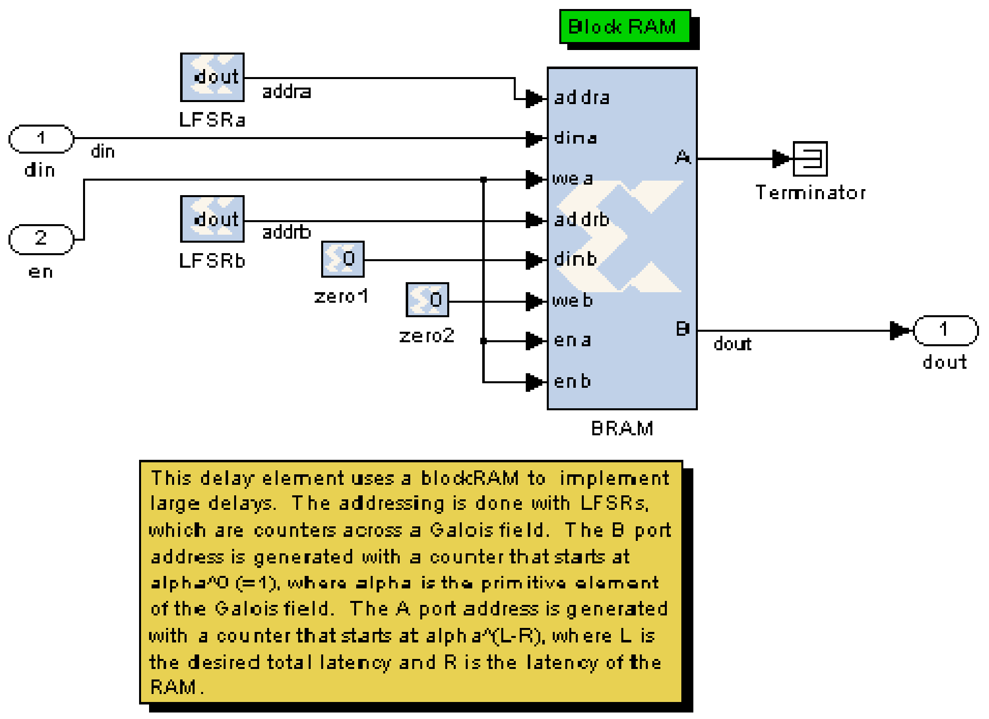

reg_array[0].has_2_latency.u1
reg_array[0].has_latency.u2
SRLC32E
FDRE
delay1
delay2
srl_delay.reg1
has_only_1.srlc33e_array0
srlc33e__parameterized0
synth_reg__parameterized0
delay_xldelay__parameterized0
srlc33e__parameterized1
synth_reg__parameterized1
delay_xldelay__parameterized1

Note that an equivalent to the faster design results from setting the latency of *Inverter1* to *1* and eliminating *Delay1*. This, however, is not equivalent to setting the latency of *Inverter1* to *4* and eliminating the delay blocks; this would yield a synthesis equivalent to the upper (slower) design.

Implementing Long Delays

For very long delays, of, say, greater than 128 cycles, especially when coupled with larger bus widths, it might be better to use a block-RAM-based delay block. The delay block is implemented using SRLs, which are part of the general fabric in the Xilinx. Very long delays should be implemented in the embedded block RAMs to save fabric. Such a delay exploits the dual-port nature of the blockRAM and can be implemented with a fixed or run-time-variable delay. Such a block is basically a block RAM with some associated address counters. The model below shows a novel way of implementing a long delay using LFSRs (linear feedback shift registers) for the address counters in order to make the design faster, but conventional counters can be used as well. The difference in value between the counters (minus the RAM latency) is the latency L of the delay line.

Figure 316: **Novel Use of Long Delay LFSRs**



This delay element uses a blockRAM to implement large delays. The addressing is done with LFSRs, which are counters across a Galois field. The B port address is generated with a counter that starts at alpha^0 (=1), where alpha is the primitive element of the Galois field. The A port address is generated with a counter that starts at alpha^(L-R), where L is the desired total latency and R is the latency of the RAM.

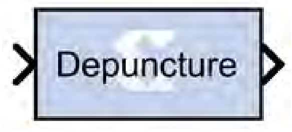
Re-settable Delays and Initial Values

If a delay line absolutely must be re-settable to zero, this can be done by using a string of L register blocks to implement the delay or by creating a circuit that forces the output to be zero while the delay line is “flushed”.

The delay block does not support initial values, but the Addressable Shift Register block does. This block, when used with a fixed address, is generally equivalent to the delay block and will synthesize to an SRL-based delay line. The initial values pertain to initialization only and not to a reset. If using the addressable shift register in “structural HDL mode” (e.g., the **Use behavioral HDL** checkbox is not selected) then the delay line will not be terminated with a flip-flop, making it significantly slower. This can be remedied by using behavioral mode or by putting a Register or Delay block after the addressable shift register.

Depuncture

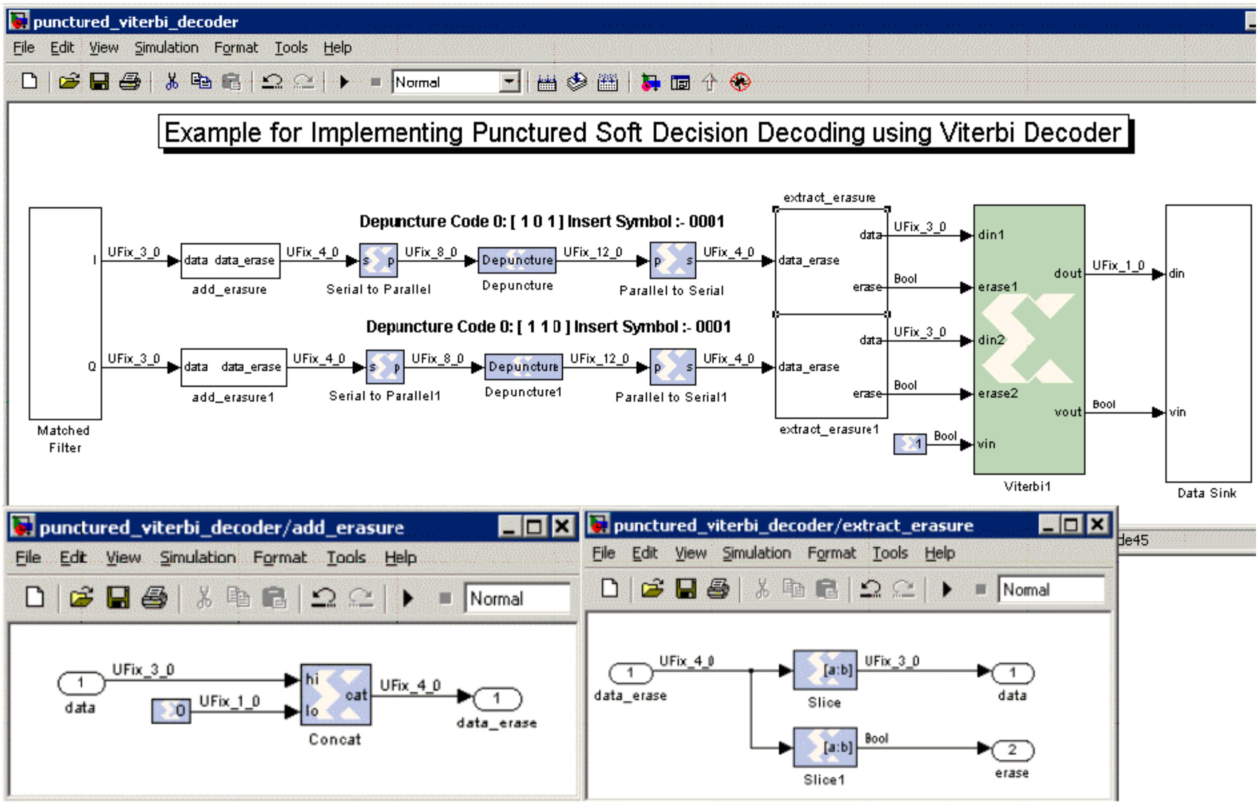
The Xilinx Depuncture block allows you to insert an arbitrary symbol into your input data at the location specified by the depuncture code.



The Xilinx depuncture block accepts data of type `UFixN_0` where N equals the length of insert string x (the number of ones in the depuncture code) and produces output data of type `UFixK_0` where K equals the length of insert string multiplied by the length of the depuncture code.

The Xilinx Depuncture block can be used to decode a range of punctured convolution codes. The following diagram illustrates an application of this block to implement soft decision Viterbi decoding of punctured convolution codes.

Figure 317: **Soft Decision Viterbi Decoding**

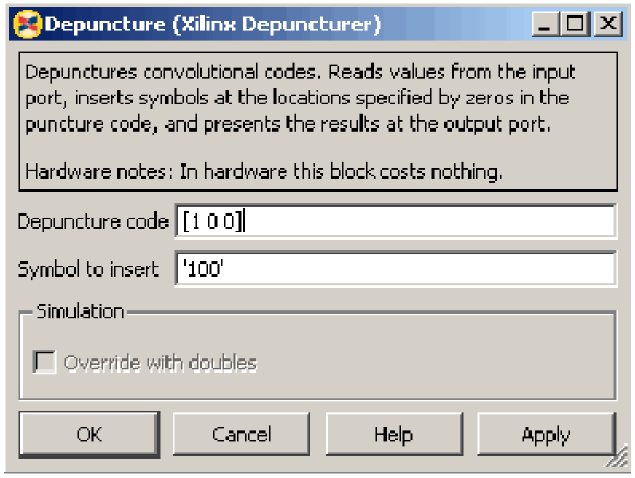

punctured_viterbi_decoder
Example for Implementing Punctured Soft Decision Decoding using Viterbi Decoder
Depuncture Code 0: [1 0 1] Insert Symbol :- 0001
Depuncture Code 0: [1 1 0] Insert Symbol :- 0001
Matched Filter
add_erasure
Serial to Parallel
Depuncture
Parallel to Serial
extract_erasure
Viterbi1
Data Sink
punctured_viterbi_decoder/add_erasure
Concat
punctured_viterbi_decoder/extract_erasure
Slice
Slice1

The previous diagram shows a matched filter block connected to a add_erasure Subsystem which attaches a 0 to the input data to mark it as a non-erasure signal. The output from the add_erasure subsystem is then passed to a serial to parallel block. The serial to parallel block concatenates two continuous soft inputs and presents it as a 8-bit word to the depuncture block. The depuncture block inserts the symbol '0001' after the 4-bits from the MSB for code 0 ([1 0 1]) and 8-bits from the MSB for code 1 ([1 1 0]) to form a 12-bit word. The output of the depuncture block is serialized as 4-bit words using the parallel to serial block. The extract_erasure Subsystem takes the input 4-bit word and extracts 3-bits from the MSB to form a soft decision input data word and 1-bit from the LSB to form the erasure signal for the Viterbi decoder.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

Figure 318: **Block Parameters**


Depuncture (Xilinx Depuncturer)

Depunctures convolutional codes. Reads values from the input port, inserts symbols at the locations specified by zeros in the puncture code, and presents the results at the output port.

Hardware notes: In hardware this block costs nothing.

Depuncture code [1 0 0]

Symbol to insert '100'

Simulation

☐ Override with doubles

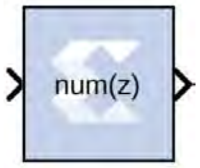
Parameters specific to the Xilinx Depuncturer block are:

- **Depuncture code:** Specifies the depuncture pattern for inserting the string to the input.
- **Symbol to insert:** Specifies the binary word to be inserted in the depuncture code.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Digital FIR Filter

The Xilinx Digital FIR Filter block allows you to generate highly parameterizable, area-efficient, high-performance single channel FIR filters.



The Digital FIR filter block supports single channel, simple rate, integer decimation, and interpolation and fractional decimation and interpolation filter types.

To specify the coefficient vector for the FIR Filter generated by this block, you can either enter the coefficient vector directly into the Digital FIR Filter block parameters dialog box, or open an interface to the FDATool block and specify the coefficient vector in that interface.

The Digital FIR Filter block is ideal for generating simple, single channel FIR filters. If your FIR filter implementation will use more complicated filter features such as multiple channels or multiple path core configuration, an AXI4-Stream-compliant interface, or functions such as reloading co-efficient, channel pattern support, or other HDL-based GUI parameters, use the Xilinx FIR Compiler 7.2 block in your design instead of the Digital FIR Filter block.

In the Vivado® design flow, the Digital FIR filter block is inferred as "LogiCORE™ IP FIR Compiler v7.2" for code generation. Refer to the document LogiCORE IP FIR Compiler v7.2 for details on this LogicCore IP.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

Parameters specific to the Xilinx Digital FIR Filter block areas follows.

- **Coefficient Vector:**
    - **Use FDA Tool as Coefficient Source: :** If selected,the Coefficient Vector will be determined by the settings in the Filter Design and Analysis Tool (FDA Tool). To use the FDA Tool as your coefficient source, you must click the **FDATool** button and configure the Block Parameters dialog box that appears, to describe your FIR filter.

      Note: Because the FDA Tool functionality is integrated into the Digital FIR Filter block itself, you do not have to enter a separate FDATool block into your design to use the FDA Tool as your coefficient source.

      The FDA Tool is a user interface for designing and analyzing filters quickly. FDATool enables you to design digital FIR filters by setting filter specifications, by importing filters from your MATLAB® workspace, or by adding, moving or deleting poles and zeroes. FDA Tool also provides tools for analyzing filters, such as magnitude and phase response and pole-zero plots (see FDATool).

    - **Edit Box:**

      The edit box is enabled for you to specify the Coefficient Vector when the **Use FDA Tool as Coefficient Source** option is disabled. The edit box specifies the vector coefficients of the filter's transfer function. Filter coefficients must be specified as a single MATLAB row vector. Filter structure must be Direct Form, and the input must be a scalar.

      The number of taps is inferred from the length of the MATLAB row vector. If multiple coefficient sets are specified, then each set is appended to the previous set in the vector.

    - **FDATool:** This button is enabled if the **Use FDA Tool as Coefficient Source** option is enabled. Click this button to open a Block Parameters dialog box for the FDA Tool, and enter your filter specifications in this dialog box. To understand how to use this dialog box to describe your FIR filter, see FDATool.

- **Coefficient Precision:**
    - **Optimal Values:** If selected, the Coefficient Width and Coefficient Fractional Bits will be set automatically to their optimum values. The values are calculated using the dynamic range of filter response between pass band and stop band signals. These values ensure the minimum hardware will be used for the required filter response when the design is implemented in the Xilinx FPGA or SoC.
    - **Coefficient Width:** Specifies the number of bits used to represent the coefficients.
    - **Coefficient Fractional Bits:** Specifies the binary point location in the coefficients datapath options.
    - **Interpolation Rate:** Specifies the interpolation rate of the filter. Any value greater than 1 is applicable to all Interpolation filter types and Decimation filter types for Fractional Rate Change implementations. The value provided in this field defines the upsampling factor, or P for Fixed Fractional Rate (P/Q) resampling filter implementations.
    - **Decimation Rate:** Specifies the decimation rate of the filter. Any value greater than 1 is applicable to the all Decimation and Interpolation filter types for Fractional Rate Change implementations. The value provided in this field defines the downsampling factor, or Q for Fixed Fractional Rate (P/Q) resampling filter implementations.

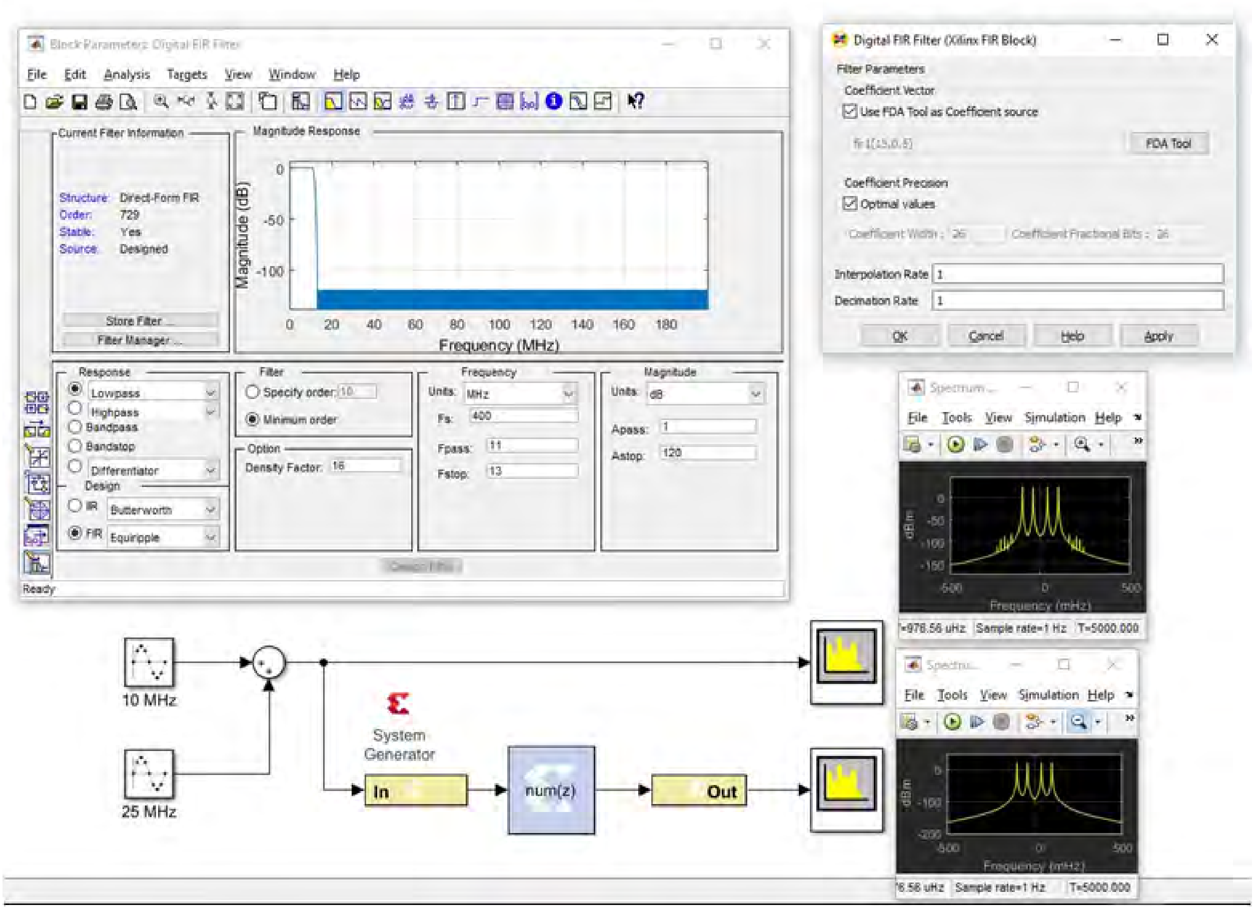
Example

A simple filter design is shown below which uses the Digital FIR Filter block to implement a single rate low pass filter. Because **Use FDA Tool as Coefficient source** is enabled in the block parameters dialog box for the Digital FIR Filter block, the FDA Tool (invoked by clicking the **FDA Tool** button) is used to generate the filter coefficient for the following specification:

- **Fs** (sample frequency) = 400 MHz
- **Fpass** = 11 MHz
- **Fstop** = 13 MHz
- **Apass** = 1 dB
- **Astop** = 120 dB

For **Coefficient precision**, the **Optimal values** selection is enabled for the filter **Coefficient Width** parameter. Therefore, an optimized filter coefficient width will be computed automatically, for minimum hardware usage and better filter response.

Figure 319: **Digital FIR Filter Example**



Current Filter Information
Structure: Direct-Form FIR
Order: 729
Stable: Yes
Source: Designed

Response: [✓ Lowpass] Highpass Bandpass Bandstop Differentiator

Design: IIR Butterworth [✓ FIR Equiripple]

Filter: Specify order: 10 [✓ Minimum order]

Option: Density Factor: 16

Frequency: Units: MHz; Fs: 400; Fpass: 11; Fstop: 13

Magnitude: Units: dB; Apass: 1; Astop: 120

Digital FIR Filter (Xilinx FIR Block)

Filter Parameters

Coefficient Vector

☑ Use FDA Tool as Coefficient source

fir1(15,0.5) FDA Tool

Coefficient Precision

☑ Optimal values

Coefficient Width : 26 Coefficient Fractional Bits : 26

Interpolation Rate 1

Decimation Rate 1

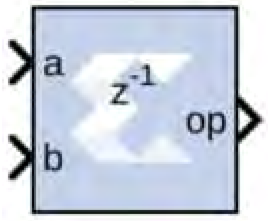
OK Cancel Help Apply

LogiCORE Documentation

LogiCORE IP FIR Compiler v7.2

Divide

The Xilinx Divide block performs both fixed-point and floating-point division with the **a** input being the dividend and the **b** input the divisor. Both inputs must be of the same data type.



Block Parameters

- **Basic tab:** Parameters specific to the Basic tab are as follows
    - **AXI Interface:**
        - **Flow Control:**
            - **Blocking:** Selects “Blocking” mode. In this mode, the lack of data on one input channel does block the execution of an operation if data is received on another input channel.
            - **NonBlocking:** Selects “Non-Blocking” mode. In this mode, the lack of data on one input channel does not block the execution of an operation if data is received on another input channel.
        - **Fixed-point Options:**
            - **Algorithm Type:**
                - **Radix2:** This is non-restoring integer division using integer operands and allows a remainder to be generated. This option is recommended for operand widths less than 16 bits. This option supports both unsigned (two's complement) and signed divisor and dividend inputs.
                - **High_Radix:** This option is recommended for operand widths greater than 16 bits, though the implementation requires the use of DSP48 (or variant) primitives. This option only supports signed (two's complement) divisor and dividend inputs.
                - **LutMult:** A simple lookup estimate of the reciprocal of the divisor followed by a multiplier. Only remainder output type is supported because of the bias required in the reciprocal estimate. This bias would introduce an offset (error) if used to create a fractional output. This is recommened for operand widths less than or equal to 12 bits.This implementation uses DSP slices, block RAM, and a small number of FPGA logic primitives (registers and LUTs). For operand widths where either Radix2 or the LUTMultoptions are possible, the LUTMult solution offers a solution using fewer FPGA logic resources because of the use of DSP and block RAM primitives. Supports unsigned or two's complement signed numbers.
            - **Output Fractional width:** For Fixed-point division, this entry determines the number of bits in the fractional part of the output.
        - **Optional Ports:**
            - **Dividend Channel Ports:**
                - **Has TLAST:** Adds a TLAST port to the Input channel.
                - **Has TUSER:** Adds a TUSER port to the Input channel.
        - **Divisor Channel Ports:**

    - **Has TLAST:** Adds a TLAST port to the Input channel.
    - **Has TUSER:** Adds a TUSER port to the Input channel.
- **Control Options:**
    - **Provide enable port:** Adds an enable port to the block interface.
    - **Has Result TREADY:** Adds a TREADY port to the Result channel.
    - **Output TLAST behavior:** Determines the behavior of the result_tlast output port.
        - **Pass_A_TLAST:** Pass the value of the a_tlast input port to the dout_tlast output port.
        - **Pass B_TLAST:** Pass the value of the b_tlast input port to the dout_tlast output port.
        - **OR_all_TLASTS:** Pass the logical OR of all the present TLAST input ports.
        - **AND_all_TLASTS:** Pass the logical AND of all the present TLAST input ports.
- **Exception Signals:**
    - **UNDERFLOW:** Adds an output port that serves as an underflow flag.
    - **OVERFLOW:** Adds an output port that serves as an overflow flag.
    - **INVALID_OP:** Adds an output port that serves as an invalid operation flag.
    - **DIVIDE_BY_ZERO:** Adds an output port that serves as a divide-by-zero flag.

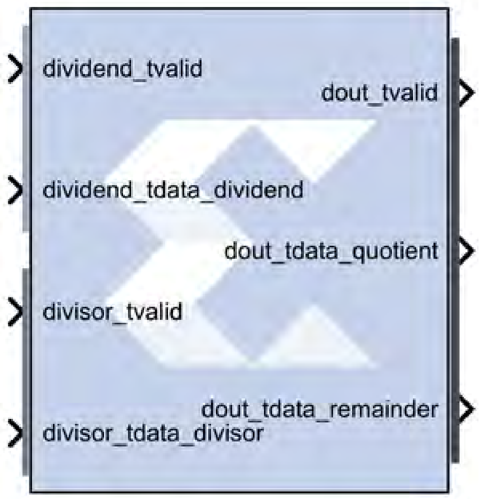
Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP Floating-Point Operator v7.1

Divider Generator 5.1

The Xilinx Divider Generator block creates a circuit for integer division based on Radix-2 non-restoring division, or High-Radix division with prescaling.



Block Parameters

The Block Parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**
  - **Common Options:**
    - **Algorithm Type:**
      - **Radix-2** non-restoring integer division using integer operands, allows a remainder to be generated. This is recommended for operand widths less than around 16 bits. This option supports both **unsigned** and **signed** (2's complement) divisor and dividend inputs.
      - **High_Radix** division with prescaling. This is recommended for operand widths greater than 16 bits, though the implementation requires the use of DSP48 (or variant) primitives. This option only supports **signed** (2's complement) divisor and dividend inputs.
      - **LutMult**A simple lookup estimate of the reciprocal of the divisor followed by a multiplier. Only the remainder output type is supported because of the bias required in the reciprocal estimate. This bias would introduce an offset (error) if used to create a fractional output. This is recommened for operand widths less than or equal to 12 bits. This implementation uses DSP slices, block RAM, and a small number of FPGA logic primitives (registers and LUTs). For operand widths where either Radix2 or the LUTMult options are possible, the LUTMult offers a solution using fewer FPGA logic resources because of the use of DSP and block RAM primitives. Supports `unsigned` or two's complement `signed` numbers.
  - **Output channel:**
    - **Remainder type:**

        - **Remainder:** Only supported for Radix 2.
        - **Fractional:** Determines the number of bits in the fractional port output.
    - **Fractional width:** If Fractional Remainder type is selected, this entry determines the number of bits in the fractional port output.
  - **Radix2 Options:**
    - **Radix2 throughput:** Determines the interval in clocks between new data being input (and output). Choices are 1, 2, 4, and 8.
  - **High Radix Options:**
    - **Detect divide by zero:** Determines if the core shall have a division-by-zero indication output port.
  - **AXI Interface:**
    - **AXI behavior:**
      - **NonBlocking:** Preforms an action only when a control packet and a data packet are presented to the block at the same time.
      - **Blocking:** Preforms an action when a data packet is presented to the block. The block uses the previous control information.
    - **AXI Implementation emphasis:**
      - **Resources:** *Automatic* (fully pipelined) or *Manual* (determined by following field).
      - **Performance:** Implementation decisions target the highest speed.
  - **Latency Options:**
    - **Latency configuration:** *Automatic* (fully pipelined) or *Manual* (determined by following field).
    - **Latency:** This field determines the exact latency from input to output in terms of clock enabled clock cycles.
- **Optional ports tab:**
  - **Optional Ports:**
    - **Divided Channel Ports:**
      - **Has TUSER:** Adds a tuser input port to the dividend channel.
      - **Has TLAST:** Adds a tlast output port to the dividend channel.
    - **Divisor Channel Ports:**

        - **Has TUSER:** Adds a tuser input port to the divisor channel.
        - **Has TLAST:** Adds a tlast output port to the divisor channel.
    - **ACLKEN:** Specifies that the block has a clock enable port (the equivalent of selecting the Has ACLKEN option in the CORE Generator GUI).
    - **ARESETn:** Specifies that the block has a reset port. Active-Low synchronous clear. A minimum ARESETn pulse of two cycles is required.
    - **m_axis_dout_tready:** Specifies that the block has a dout_tready output port.
- **Input TLAST combination for output:** Determines the behavior of the dout_tlast output port.
    - **Null:** Output is null.
    - **Pass_Dividend_TLAST:** Pass the value of the dividend_tlast input port to the dout_tlast output port.
    - **Pass Divisor_TLAST:** Pass the value of the divisor_tlast input port to the dout_tlast output port.
    - **OR_all_TLASTS:** Pass the logical OR of all the present TLAST input ports.
    - **AND_all_TLASTS:** Pass the logical AND of all the present TLAST input ports.

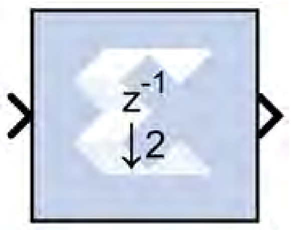
Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP Divider Generator 5.1

Down Sample

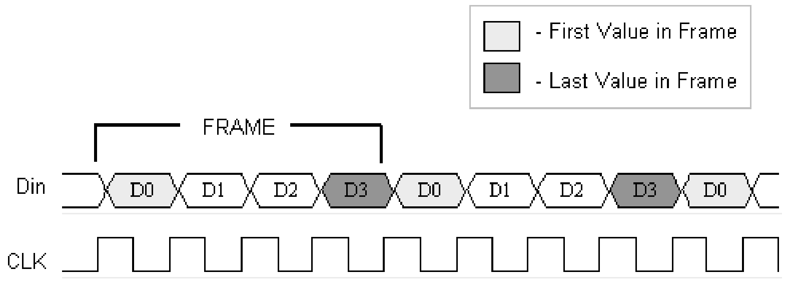
The Xilinx Down Sample block reduces the sample rate at the point where the block is placed in your design.



The input signal is sampled at even intervals, at either the beginning (first value), or end (last value) of a frame. The sampled value is presented on the output port and held until the next sample is taken.

A Down Sample frame consists of I input samples, where I is sampling rate. An example frame for a Down Sample block configured with a sampling rate of 4 is shown below.

Figure 320: **Down Sample Block Example**

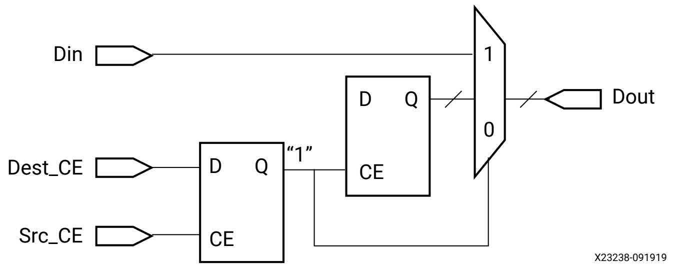


The Down Sample block is realized in hardware using one of three possible implementations that vary in terms of implementation efficiency. The block receives two clock enable signals in hardware, Src_CE, and Dest_CE. Src_CE is the faster clock enable signal and corresponds to the input data stream rate. Dest_CE is the slower clock enable, corresponding to the output stream rate, for example, down sampled data. These enable signals control the register sampling in hardware.

Zero Latency Down Sample

The zero latency Down Sample block must be configured to sample the first value of the frame. The first sample in the input frame passes through the mux to the output port. A register samples this value during the first sample duration and the mux switches to the register output at the start of the second sample of the frame. The result is that the first sample in a frame is present on the output port for the entire frame duration. This is the least efficient hardware implementation as the mux introduces a combinational path from Din to Dout. A single bit register adjusts the timing of the destination clock enable, so that it is asserted at the start of the sample period, instead of the end. The hardware implementation is shown below:

Figure 321: **Down Sample with Zero Latency Example**

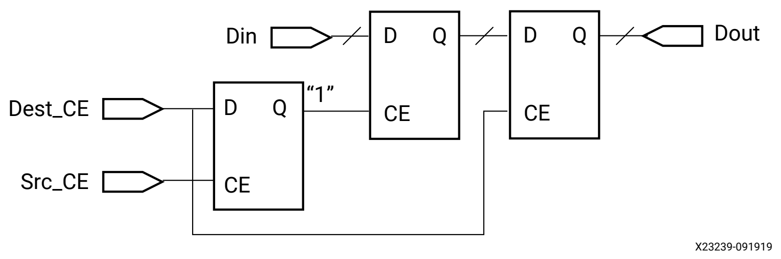


Down Sample with Latency

If the Down Sample block is configured with latency greater than zero, a more efficient implementation is used. One of two implementations is selected depending on whether the Down Sample block is set to sample the first or last value in a frame.

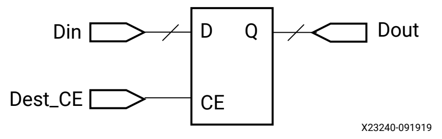
If the block samples the first value in a frame, two registers are required to correctly sample the input stream. The first register is enabled by the adjusted clock enable signal so that it samples the input at the start of the input frame. The second register samples the contents of the first register at the end of the sample period to ensure output data is aligned correctly.

Figure 322: **Down Sample with Latency Example**



If the block samples the last value in a frame, a register samples the data input data at the end of the frame. The sampled value is presented for the duration of the next frame. The most efficient implementation is when the Down Sample block is configured to sample the last value of the frame.

Figure 323: **Sampling Last Value**



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**
  - **Sampling Rate (number of input samples per output sample):** Must be an integer greater or equal to 2. This is the ratio of the output sample period to the input, and is essentially a sample rate divider. For example, a ratio of 2 indicates a 2:1 division of the input sample rate. If a non-integer ratio is desired, the Up Sample block can be used in combination with the Down Sample block.
  - **Sample:** The Down Sample block can sample either the first or last value of a frame. This parameter will determine which of these two values is sampled.

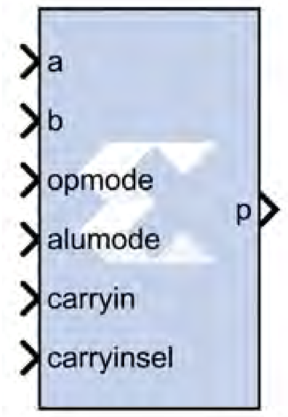
Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Xilinx LogiCORE

The Down Sample block does not use a Xilinx LogiCORE.

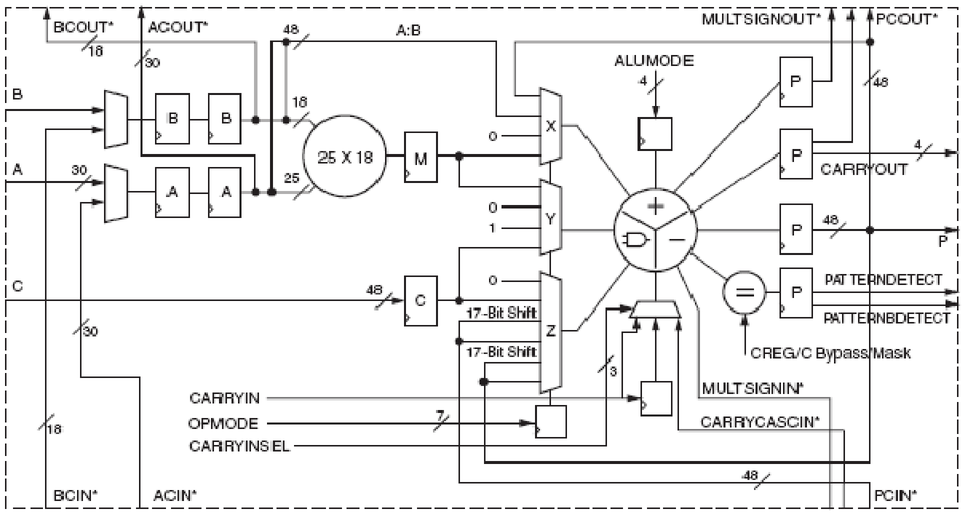
DSP48E

The Xilinx DSP48E block is an efficient building block for DSP applications that use supported devices. The DSP48E combines an 18-bit by 25-bit signed multiplier with a 48-bit adder and programmable mux to select the adder's input.



Operations can be selected dynamically. Optional input and multiplier pipeline registers can be selected as well as registers for the alumode, carryin and opmode ports. The DSP48E block can also target devices that do not contain the DSP48E hardware primitive if the **Use synthesizable** model option is selected on the implementation tab.

Figure 324: **DSP48E**



*These signals are dedicated routing paths internal to the DSP48E column. They are not accessible via fabric routing resources.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**
  - **A or ACIN input:** Specifies if the A input should be taken directly from the a port or from the cascaded acin port. The acin port can only be connected to another DSP48 block.
  - **B or BCIN input:** Specifies if the B input should be taken directly from the b port or from the cascaded bcin port. The bcin port can only be connected to another DSP48 block.
- **Pattern Detection:**
  - **Reset p register on pattern detection:** If selected and the pattern is detected, reset the p register on the next cycle
  - **Pattern Input:**
    - **Pattern Input from c port:** When selected, the pattern used in pattern detection is read from the c port.
    - **Using Pattern Attribute (48bit hex value):** Value is used in pattern detection logic which is best described as an equality check on the output of the adder/subtractor/logic unit.
    - **Pattern attribute:** A 48-bit value that is used in the pattern detector.
  - **Mask Input:**
    - **Mask input from c port:** When selected, the mask used in pattern detection is read from the c port.

        - **Using Mask Attribute (48 bit hex value):** 48-bit value used to mask out certain bits during pattern detection.
        - **Mask attribute:** A 48-bit value and used to mask out certain bits during a pattern detection. A value of 0 passes the bit, and a value of 1 masks out the bit.48-bit value and used to mask out certain bits during a pattern detection.
    - **Select rounding mask:** Selects special masks that can be used for symmetric or convergent rounding in the pattern detector. The choices are **Select mask**, **Mode1**, and **Mode2**.
- **Optional Ports tab:**
    - **Input Ports:**
        - **Consolidate control port:** When selected, combines the opmode, alumode, carry_in and carry_in_sel ports into one 15-bit port. Bits 0 to 6 are the opmode, bits 7 to 10 are the alumode port, bit 11 is the carry_in port, and bits 12 to 14 are the carry_in_sel port. This option should be used when the Opmode block is used to generate a DSP48E instruction.
        - **Provide c port:** When selected, the c port is made available. Otherwise, the c port is tied to '0'.
        - **Provide global reset port:** When selected, the port rst is made available. This port is connected to all available reset ports based on the pipeline selections.
        - **Provide global enable port:** When selected, the optional en port is made available. This port is connected to all available enable ports based on the pipeline selections.
    - **Cascadable Ports:**
        - **Provide pcin port:** When selected, the pcin port is exposed. The pcin port must be connected to the pcout port of another DSP48 block.
        - **Provide carry cascade in port:** When selected, the carry cascade in port is exposed. This port can only be connected to a carry cascade out port on another DSP48E block.
        - **Provide multiplier sign cascade in port:** When selected, the multiplier sign cascade in port (multsigncascin) is exposed. This port can only be connected to a multiplier sign cascade out port of another DSP48E block.
    - **Output Ports:**
        - **Provide carryout port:** When selected, the carryout output port is made available. When the mode of operation for the adder/subtractor is set to one 48-bit adder, the carryout port is 1-bit wide. When the mode of operation is set to two 24 bit adders, the carryout port is 2 bits wide. The MSB corresponds to the second adder's carryout and the LSB corresponds to the first adder's carryout. When the mode of operation is set to four 12 bit adders, the carryout port is 4 bits wide with the bits corresponding to the addition of the 48 bit input split into 4 12-bit sections.

- **Provide pattern detect port:** When selected, the pattern detection output port is provided. When the pattern, either from the mask or the c register, is matched the pattern detection port is set to '1'.
- **Provide pattern bar detect port:** When selected, the pattern bar detection (patternbdetect) output port is provided. When the inverse of the pattern, either from the mask or the c register, is matched the pattern bar detection port is set to '1'.
- **Provide overflow port:** When selected, the overflow output port is provided. This port indicates when the operation in the DSP48E has overflowed beyond the bit P[N] where N is between 1 and 46. N is determined by the number of 1s in the mask whether set by the GUI mask field or the c port input.
- **Provide underflow port:** When selected, the underflow output port is provided. This port indicates when the operation in the DSP48E has underflowed. Underflow occurs when the number goes below –P[N] where N is determined by the number of 1s in the mask whether set by the GUI mask field or the c port input.

- **Cascadable Ports:**
  - **Provide ACOUT port:** When selected, the acout output port is made available. The acout port must be connected to the acin port of another DSP48E block.
  - **Provide BCOUT port:** When selected, the bcout output port is made available. The bcout port must be connected to the bcin port of another DSP48E block.
  - **Provide PCOUT port:** when selected, the pcout output port is made available. The pcout port must be connected to the pcin port of another DSP48 block.
  - **Provide multiplier sign cascade out port:** When selected, the multiplier sign cascade out port (multsigncascout) is made available. This port can only be connected to the multiplier sign cascade in port of another DSP48E block and is used to support 96-bit accumulators/adders and subtracters which are built from two DSP48Es.
  - **Provide carry cascade out port:** When selected, the carry cascade out port (carrycascout) is made available. This port can only be connected to the carry cascade in port of another DSP48E block.

- **Pipelining tab:**
  - **Pipeline Options:**
    - **Length of a/acin pipeline:** Specifies the length of the pipeline on input register A. A pipeline of length 0 removes the register on the input.
    - **Length of b/bCIN pipeline:** Specifies the length of the pipeline for the b input whether it is read from b or bcin.
    - **Length of acout pipeline:** Specifies the length of the pipeline between the a/acin input and the acout output port. A pipeline of length 0 removes the register from the acout pipeline length. Must be less than or equal to the length of the a/acin pipeline.

    - **Length of bcout pipeline:** Specifies the length of the pipeline between the b/bcin input and the bcout output port. A pipeline of length 0 removes the register from the bcout pipeline length. Must be less than or equal to the length of the b/bcin pipeline.
    - **Pipeline c:** Indicates whether the input from the c port should be registered.
    - **Pipeline p:** Indicates whether the outputs p and pcout should be registered.
    - **Pipeline multiplier:** indicates whether the internal multiplier should register its output.
    - **Pipeline opmode:** Indicates whether the opmode port should be registered.
    - **Pipeline alumode:** Indicates whether the alumode port should be registered.
    - **Pipeline carry in:** Indicates whether the carry in port should be registered.
    - **Pipeline carry in select:** Indicates whether the carry in select port should be registered.

- **Reset/Enable Ports tab:**
  - **Reset port for a/acin:** When selected, a port rst_a is made available. This resets the pipeline register for port a when set to '1'.
  - **Reset port for b/bcin:** When selected, a port rst_b is made available. This resets the pipeline register for port b when set to '1'.
  - **Reset port for c:** When selected, a port rst_c is made available. This resets the pipeline register for port c when set to '1'.
  - **Reset port for multiplier:** when selected, a port rst_m is made available. This resets the pipeline register for the internal multiplier when set to '1'.
  - **Reset port for P:** When selected, a port rst_p is made available. This resets the output register when set to '1'.
  - **Reset port for carry in:** When selected, a port rst_carryin is made available. This resets the pipeline register for carry in when set to '1'.
  - **Reset port for alumode:** When selected, a port rst_alumode is made available. This resets the pipeline register for the alumode port when set to '1'.
  - **Reset port for controls (opmode and carry_in_sel):** When selected, a port rst_ctrl is made available. This resets the pipeline register for the opmode register (if available) and the carry_in_sel register (if available) when set to '1'.
  - **Enable port for first a/acin register:** When selected, an enable port ce_a1 for the first a pipeline register is made available.
  - **Enable port for second a/acin register:** When selected, an enable port ce_a2 for the second a pipeline register is made available.
  - **Enable port for first b/bcin register:** When selected, an enable port ce_b1 for the first b pipeline register is made available.

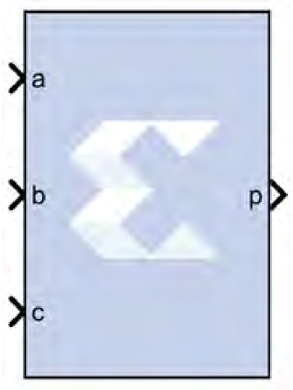
- **Enable port for second b/bcin register:** When selected, an enable port ce_b2 for the second b pipeline register is made available.
- **Enable port for c:** When selected, an enable port ce_c for the port C register is made available.
- **Enable port for multiplier:** When selected, an enable port ce_m for the multiplier register is made available.
- **Enable port for p:** When selected, an enable port ce_p for the port P output register is made available.
- **Enable port for carry in:** When selected, an enable port ce_carry_in for the carry in register is made available.
- **Enable port for alumode:** When selected, an enable port ce_alumode for the alumode register is made available.
- **Enable port for multiplier carry in:** When selected, an enable port mult_carry_in for the multiplier register is made available.
- **Enable port for controls (opmode and carry_in_sel):** When selected, the enable port ce_ctrl is made available. The port ce_ctrl controls the opmode and carry in select registers.

- **Implementation tab:**
  - **Use synthesizable model:** When selected, the DSP48E is implemented from an RTL description which might not map directly to the DSP48E hardware. This is useful if a design using the DSP48E block is targeted at device families that do not contain DSP48E hardware primitives.
  - **Mode of operation for the adder/subtractor:** This mode can be used to implement small add-subtract functions at high speed and lower power with less logic utilization. The adder and subtracter in the adder/subtracted/logic unit can also be split into two 24-bit fields or four12-bit fields. This is achieved by setting the mode of operation to "Two 24-bit adders" or "Four 12-bit adders".
  - **Use adder only:** When selected, the block is optimized in hardware for maximum performance without using the multiplier. If an instruction using the multiplier is encountered in simulation, an error is reported.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

DSP Macro 1.0

The Xilinx DSP macro block provides a device independent abstraction of the DSP48E1, DSP48E2, and DSP58 blocks. Using this block instead of using a technology-specific DSP slice helps makes the design more portable between Xilinx technologies.



The DSP Macro provides a simplified interface to the XtremeDSP slice by the abstraction of all opmode, subtract, alumode, and inmode controls to a single SEL port. Further, all CE and RST controls are grouped to a single CE and SCLR port respectively. This abstraction enhances portability of HDL between device families.

You can specify 1 to 64 instructions which are translated into the various control signals for the XtremeDSP slice of the target device. The instructions are stored in a ROM from which the appropriate instruction is selected using the SEL port.

Block Parameters

- **Instruction tab:**

  The Instruction tab is used to define the operations that the LogiCORE™ is to implement. Each instruction can be entered on a new line, or in a comma delimited list, and are enumerated from the top down. You can specify a maximum of 64 instructions.

  Refer to the topic Instructions page of the LogiCORE IP DSP Macro 1.0 Product Guide for details on all the parameters on this tab.

- **Pipeline Options tab:**

  The Pipeline Options tab is used to define the pipeline depth of the various input paths.

  - **Pipeline Options:**

    Specifies the pipeline method to be used; **Automatic**, **By Tier**, or **Expert**.

  - **Custom Pipeline options:**

    Used to specify the pipeline depth of the various input paths.

  - **Tier 1 to 6:** When **By Tier** is selected for Pipeline Options these parameters are used to enable/disable the registers across all the input paths for a given pipeline stage. The following restrictions are enforced:

        - When P has been specified in an expression tier, 6 will be forced as asynchronous feedback is not supported.

    - **Individual registers:**

      When you select **Expert** for the Pipeline Options, these parameters are used to enable/disable individual register stages. The following restrictions are enforced:

        - The P register is forced when P is specified in an expression. Asynchronous feedback is not supported.

  Refer to the topic Detailed Pipeline Implementation of the LogiCORE IP DSP Macro v1.0 Product Guide for details on all the parameters on this tab.

- **Implementation tab:**

  The Implementation tab is used to define implementation options.

    - **Output Port Properties:**

        - **Precision:** Specifies the precision of the P output port.

            - **Full:** The bit width of the output port P is set to the full XtremeDSP Slide width of 48 bits.

            - **User_Defined:** The output width of P can be set to any value up to 48 bits. When set to less than 48 bits, the output is truncated (LSBs removed).

        - **Width:** Specifies the User Defined output width of the P output port

        - **Binary Point:** Specifies the placement of the binary point of the P output port.

    - **Additional ports:**

        - **Use ACOUT:** Use the optional cascade A output port.

        - **Use CARRYOUT:** Use the optional carryout output port.

        - **Use BCOUT:** Use the optional cascade B output port.

        - **Use CARRYCASCOUT:** Use the optional cascade carryout output port.

        - **Use PCOUT:** Use the optional cascade P output port.

    - **Control ports:**

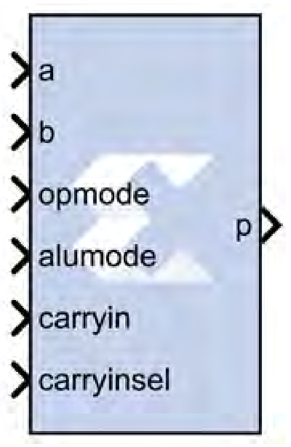
      Refer to the topic Implementation Page of the LogiCORE IP DSP Macro v1.0 Product Guide for details on all the parameters on this tab.

LogiCORE Documentation

LogiCORE IP DSP Macro v1.0 Product Guide

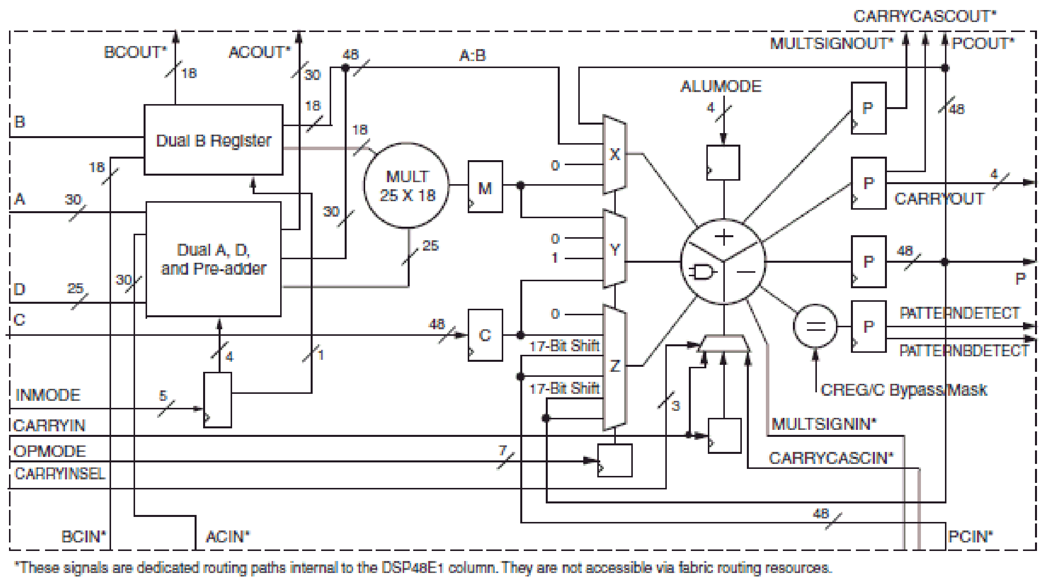
DSP48E1

The Xilinx DSP48E1 block is an efficient building block for DSP applications that use Xilinx Virtex®-7 series devices. Enhancements to the DSP48E1 slice provide improved flexibility and utilization, improved efficiency of applications, reduced overall power consumption, and increased maximum frequency. The high performance allows designers to implement multiple slower operations in a single DSP48E1 slice using time-multiplexing methods.



The DSP48E1 slice supports many independent functions. These functions include multiply, multiply accumulate (MACC), multiply add, three-input add, barrel shift, wide-bus multiplexing, magnitude comparator, bit-wise logic functions, pattern detect, and wide counter. The architecture also supports cascading multiple DSP48E1 slices to form wide math functions, DSP filters, and complex arithmetic without the use of general FPGA.

Figure 325: **DSP48E1**



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows.

  - **Input configuration:**

    - **A or ACIN input:** Specifies if the A input should be taken directly from the a port or from the cascaded `acin` port. The `acin` port can only be connected to another DSP48 block.

    - **B or BCIN input:** Specifies if the B input should be taken directly from the b port or from the cascaded `bcin` port. The `bcin` port can only be connected to another DSP48 block.

    Note: If the input to the block is NaN, you will see a behavioral simulation mismatch.

  - **DSP48E1 data-path configuration:**

    - **SIMD Mode of Adder/Subtractor/Accumulator:** This mode can be used to implement small add-subtract functions at high speed and lower power with less logic utilization. The adder and subtracter in the adder/subtracter/logic unit can also be split into **Two 24-bit Units** or **Four 12-bit Units**.

    - **Do not use multiplier:** When selected, the block is optimized in hardware for maximum performance without using the multiplier. If an instruction using the multiplier is encountered in simulation, an error is reported.

- **Use dynamic multiplier mode:** When selected, it instructs the block to use the dynamic multiplier mode. This indicates that the block is switching between A*B and A:B operations on the fly and therefore needs to get the worst-case timing of the two paths.
- **Use Preadder:** Use the 25-bit D data input to the pre-adder or alternative input to the multiplier. The pre-adder implements D + A as determined by the INMODE3 signal.

- **Pattern Detection:**
  - **Reset p register on pattern detection:** If selected and the pattern is detected, reset the p register on the next cycle
- **Pattern Input:**
  - **Pattern Input from c port:** When selected, the pattern used in pattern detection is read from the c port.
    - **Pattern Input:**
      - **Pattern Input from c port:** When selected, the pattern used in pattern detection is read from the c port.
      - **Using Pattern Attribute (48bit hex value):** Value is used in pattern detection logic which is best described as an equality check on the output of the adder/ subtracter/logic unit
      - **Pattern attribute:** A 48-bit value that is used in the pattern detector.
    - **Mask Input:**
      - **Mask input from c port:** When selected, the mask used in pattern detection is read from the c port.
      - **Using Mask Attribute (48 bit hex value):** Enter a 48-bit value used to mask out certain bits during pattern detection.
      - **MODE1:** Selects rounding_mode 1.
      - **MODE2:** Selects rounding_mode 2.

- **Optional Ports tab:**

  Parameters specific to the Optional Ports tab are:

  - **Input Ports:**
    - **Consolidate control port:** When selected, combines the `opmode`, `alumode`, `carry_in`, `carry_in_sel`, and `inmode` ports into one 20-bit port. Bits 0 to 6 are the opmode, bits 7 to 10 are the `alumode` port, bit 11 is the `carry_in` port, bits 12 to 14 are the `carry_in_sel` port, and bits 15-19 are the inmode bits. This option should be used when the Opmode block is used to generate a DSP48 instruction.

- **Provide c port:** When selected, the c port is made available. Otherwise, the c port is tied to '0'.
- **Provide global reset port:** When selected, the port rst is made available. This port is connected to all available reset ports based on the pipeline selections.
- **Provide global enable port:** When selected, the optional en port is made available. This port is connected to all available enable ports based on the pipeline selections.
- **Provide pcin port:** When selected, the `pcin` port is exposed. The `pcin` port must be connected to the `pcout` port of another DSP48 block.
- **Provide carry cascade in port:** When selected, the carry cascade in port is exposed. This port can only be connected to a carry cascade out port on another DSP48E block.
- **Provide multiplier sign cascade in port:** When selected, the multiplier sign cascade in port (multsigncascin) is exposed. This port can only be connected to a multiplier sign cascade out port of another DSP48E block.
- **Provide carryout port:** When selected, the `carryout` output port is made available. When the mode of operation for the adder/subtractor is set to one 48-bit adder, the `carryout` port is 1-bit wide. When the mode of operation is set to two 24 bit adders, the `carryout` port is 2 bits wide. The MSB corresponds to the second adder's carryout and the LSB corresponds to the first adder's carryout. When the mode of operation is set to four 12 bit adders, the `carryout` port is 4 bits wide with the bits corresponding to the addition of the 48 bit input split into 4 12-bit sections.
- **Provide pattern detect port:** When selected, the pattern detection output port is provided. When the pattern, either from the mask or the c register, is matched the pattern detection port is set to '1'.
- **Provide pattern bar detect port:** When selected, the pattern bar detection (patternbdetect) output port is provided. When the inverse of the pattern, either from the mask or the c register, is matched the pattern bar detection port is set to '1'.
- **Provide overflow port:** When selected, the overflow output port is provided. This port indicates when the operation in the DSP48E has overflowed beyond the bit P[N] where N is between 1 and 46. N is determined by the number of 1s in the mask whether set by the GUI mask field or the c port input.
- **Provide underflow port:** When selected, the underflow output port is provided. This port indicates when the operation in the DSP48E has underflowed. Underflow occurs when the number goes below –P[N] where N is determined by the number of 1s in the mask whether set by the GUI mask field or the c port input.
- **Provide acout port:** When selected, the `acout` output port is made available. The `acout` port must be connected to the `acin` port of another DSP48E block.
- **Provide bcout port:** When selected, the `bcout` output port is made available. The `bcout` port must be connected to the `bcin` port of another DSP48E block.

- **Provide pcout port:** When selected, the `pcout` output port is made available. The `pcout` port must be connected to the `pcin` port of another DSP48 block.
- **Provide multiplier sign cascade out port:** When selected, the multiplier sign cascade out port (multsigncascout) is made available. This port can only be connected to the multiplier sign cascade in port of another DSP48E block and is used to support 96-bit accumulators/adders and subtracters which are built from two DSP48Es.
- **Provide carry cascade out port:** When selected, the carry cascade out port (`carrycascout`) is made available. This port can only be connected to the carry cascade in port of another DSP48E block.

- **Pipelining tab:**

  Parameters specific to the Pipelining tab are as follows.

  - **Length of a/acin pipeline:** Specifies the length of the pipeline on input register A. A pipeline of length 0 removes the register on the input.
  - **Length of b/bcin pipeline:** Specifies the length of the pipeline for the b input whether it is read from b or `bcin`.
  - **Length of acout pipeline:** Specifies the length of the pipeline between the a/`acin` input and the `acout` output port. A pipeline of length 0 removes the register from the `acout` pipeline length. Must be less than or equal to the length of the a/`acin` pipeline.
  - **Length of bcout pipeline:** Specifies the length of the pipeline between the b/`bcin` input and the bcout output port. A pipeline of length 0 removes the register from the bcout pipeline length. Must be less than or equal to the length of the b/`bcin` pipeline.
  - **Pipeline c:** Indicates whether the input from the c port should be registered.
  - **Pipeline p:** Indicates whether the outputs p and pcout should be registered.
  - **Pipeline multiplier:** Indicates whether the internal multiplier should register its output.
  - **Pipeline opmode:** Indicates whether the opmode port should be registered.
  - **Pipeline alumode:** Indicates whether the alumode port should be registered.
  - **Pipeline carry in:** Indicates whether the carry in port should be registered.
  - **Pipeline carry in select:** Indicates whether the carry in select port should be registered.
  - **Pipeline preadder input register d:** Indicates to add a pipeline register to the d input.
  - **Pipeline preadder output register ad:** Indicates to add a pipeline register to the ad output.
  - **Pipeline INMODE register:** Indicates to add a pipeline register to the INMODE input.

- **Reset/Enable Ports:** Parameters specific to the Reset/Enable tab are as follows.

- **Provide Reset Ports:**
    - **Reset port for a/acin:** When selected, a port rst_a is made available. This resets the pipeline register for port a when set to '1'.
    - **Reset port for b/bcin:** When selected, a port rst_b is made available. This resets the pipeline register for port b when set to '1'.
    - **Reset port for c:** When selected, a port rst_c is made available. This resets the pipeline register for port c when set to '1'.
    - **Reset port for multiplier:** When selected, a port rst_m is made available. This resets the pipeline register for the internal multiplier when set to '1'.
    - **Reset port for P:** When selected, a port rst_p is made available. This resets the output register when set to '1'.
    - **Reset port for carry in:** When selected, a port rst_carryin is made available. This resets the pipeline register for carry in when set to '1'.
    - **Reset port for alumode:** When selected, a port rst_alumode is made available. This resets the pipeline register for the alumode port when set to '1'.
    - **Reset port for controls (opmode and carry_in_sel):** When selected, a port rst_ctrl is made available. This resets the pipeline register for the opmode register (if available) and the carry_in_sel register (if available) when set to '1'.
    - **Reset port for d and ad:**
    - **Reset port for INMODE:**
- **Provide Enable Ports:**
    - **Enable port for first a/acin register:** When selected, an enable port ce_a1 for the first a pipeline register is made available.
    - **Enable port for second a/acin register:** When selected, an enable port ce_a2 for the second a pipeline register is made available.
    - **Enable port for first b/bcin register:** When selected, an enable port ce_b1 for the first b pipeline register is made available.
    - **Enable port for second b/bcin register:** When selected, an enable port ce_b2 for the second b pipeline register is made available.
    - **Enable port for c:** When selected, an enable port ce_c for the port C register is made available.
    - **Enable port for multiplier:** When selected, an enable port ce_m for the multiplier register is made available.

  - **Enable port for p:** When selected, an enable port ce_p for the port P output register is made available.
  - **Enable port for carry in:** When selected, an enable port ce_carry_in for the carry in register is made available.
  - **Enable port for alumode:** When selected, an enable port ce_alumode for the alumode register is made available.
  - **Enable port for multiplier carry in:** When selected, an enable port mult_carry_in for the multiplier register is made available.
  - **Enable port for controls (opmode and carry_in_sel):** When selected, the enable port ce_ctrl is made available. The port ce_ctrl controls the opmode and carry in select registers.
  - **Enable port for d:** When selected, an enable port is added input register d.
  - **Enable port for ad:** When selected, an enable port is add for the preadder output register ad.
  - **Enable port for INMODE:** When selected, an enable port is added for the INMODE register.

- **Implementation:**

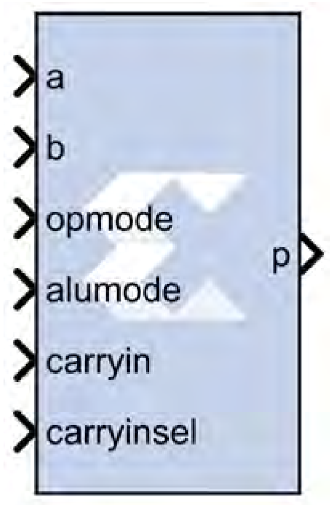
  Parameters specific to the Implementation tab are as follows.

  - **Use synthesizable model:** When selected, the DSP48E is implemented from an RTL description which might not map directly to the DSP48E hardware. This is useful if a design using the DSP48E block is targeted at device families that do not contain DSP48E hardware primitives.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

DSP48E2

The Xilinx DSP48E2 block is an efficient building block for DSP applications that use UltraScale devices. DSP applications use many binary multipliers and accumulators that are best implemented in dedicated DSP resources. UltraScale devices have many dedicated low-power DSP slices, combining high speed with small size while retaining system design flexibility.

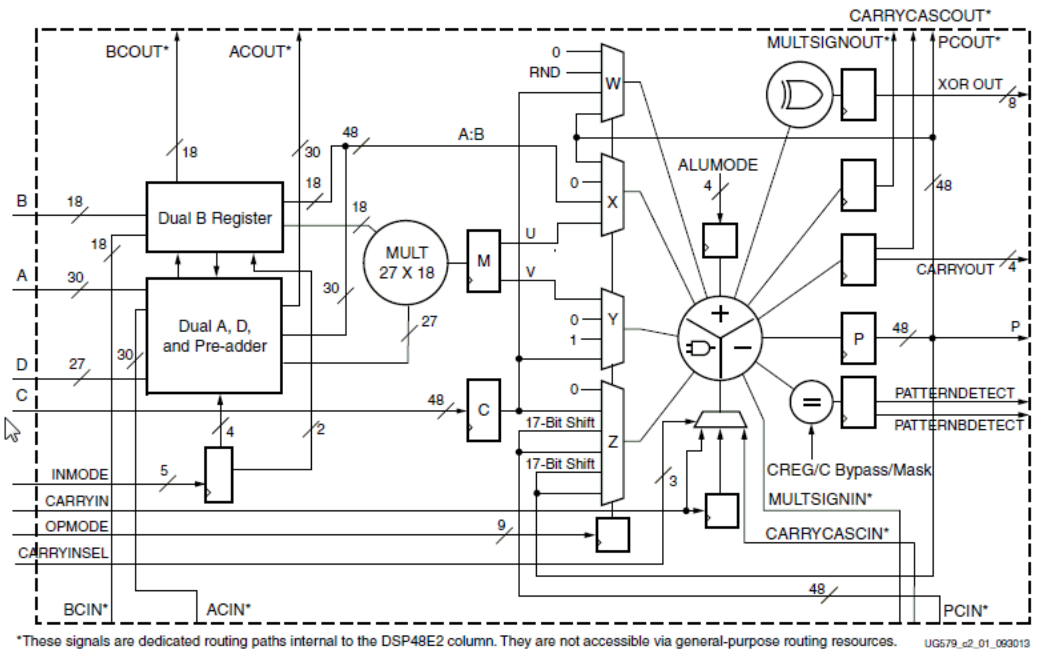


The DSP48E2 slice is effectively a superset of the DSP48E1 slice with these differences:

- Wider functionality
- More flexibility in the pre-adder
- Added fourth operand to ALU with WMUX
- Wide XOR of the X, Y, and Z multiplexers
- Additional unique features

Refer to the document titled *UltraScale Architecture DSP Slice User Guide* (UG579) for a detailed description of the DSP48E2 features.

Figure 326: **DSP48E2**



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:** Parameters specific to the Basic tab are as follows.
  - **Input configuration:**
    - **A or ACIN input:** Specifies if the A input should be taken directly from the a port or from the cascaded `acin` port. The `acin` port can only be connected to another DSP48 block.
    - **B or BCIN input:** Specifies if the B input should be taken directly from the b port or from the cascaded `bcin` port. The `bcin` port can only be connected to another DSP48 block.

    Note: If the input to the block is NaN, you will see a behavioral simulation mismatch.
  - **DSP48E2 data-path configuration:**
    - **SIMD Mode of Adder/Subtractor/Accumulator:** This mode can be used to implement small add-subtract functions at high speed and lower power with less logic utilization. The adder and subtracter in the adder/subtracted/logic unit can also be split into **Two 24-bit Units** or **Four 12-bit Units**.
    - **Do not use multiplier:** When selected, the block is optimized in hardware for maximum performance without using the multiplier. If an instruction using the multiplier is encountered in simulation, an error is reported.

    - **Use dynamic multiplier mode:** When selected, it instructs the block to use the dynamic multiplier mode. This indicates that the block is switching between A*B and A:B operations on the fly and therefore needs to get the worst-case timing of the two paths.

  - **Preadder configuration:** Use the 25-bit D data input to the pre-adder or alternative input to the multiplier. The pre-adder implements D + A as determined by the INMODE3 signal.

    - **PREADDINSEL Select preadder input:** Selects the input to be added with D in the preadder.

    - **AMULTSEL Select A multiplexer output:** Selects the input to the 27-bit A input of the multiplier. In the 7 series primitive DSP48E1 the attribute is called USE_DPORT, but has been renamed due to new pre-adder flexibility enhancements (default AMULTSEL = A is equivalent to USE_DPORT=FALSE).

    - **BMULTSEL Select B multiplexer output:** Selects the input to the 18-bit B input of the multiplier.

    - **Enable D Port:** Automatically enabled when AD is selected above.

  - **Pattern Detection:**

    - **Reset p register on pattern detection:** If selected and the pattern is detected, reset the p register on the next cycle.

    - **AUTO RESET PRIORITY:** When enabled by selecting the option above, select RESET (the default) or CEP (clock enabled for the P (output) resister).

  - **Pattern Input:**

    - **Pattern Input from c port:** When selected, the pattern used in pattern detection is read from the c port.

    - **Using Pattern Attribute (48bit hex value):** Value is used in pattern detection logic which is best described as an equality check on the output of the adder/subtractor/logic unit.

    - **Pattern Attribute (48bit hex value):** Enter a 48-bit value that is used in the pattern detector.

  - **Mask Input:**

    - **Mask input from c port:** When selected, the mask used in pattern detection is read from the c port.

    - **Using Mask Attribute (48 bit hex value):** Enter a 48-bit value used to mask out certain bits during pattern detection.

    - **MODE1:** Selects rounding_mode 1 (C-bar left shifted by 1).

    - **MODE2:** Selects rounding_mode 2 (C-bar left shifted by 2).

- **Wide Xor tab:** Parameters specific to the Wide Xor tab are as follows.

    - **Use Wide XOR:** This is a new feature in the DSP48E2 slice giving the ability to perform a 96-bit wide XOR function.
    - **XORSIMD Select Wide XOR SIMD:** The XORSIMD attribute is used to select the width of the XOR function. Select either XOR12 (the default), XOR24, XOR48, or XOR96.
- **Optional Ports tab:** Parameters specific to the Optional Ports tab are as follows.
    - **Input Ports:**
        - **Consolidate control port:** When selected, combines the `opmode`, `alumode`, `carry_in`, `carry_in_sel`, and `inmode` ports into one 20-bit port. Bits 0 to 6 are the opmode, bits 7 to 10 are the `alumode` port, bit 11 is the `carry_in` port, bits 12 to 14 are the `carry_in_sel` port, and bits 15-19 are the inmode bits. This option should be used when the Opmode block is used to generate a DSP48 instruction.
        - **Provide c port:** When selected, the c port is made available. Otherwise, the c port is tied to '0'.
        - **Provide global reset port:** When selected, the port rst is made available. This port is connected to all available reset ports based on the pipeline selections.
        - **Provide global enable port:** When selected, the optional en port is made available. This port is connected to all available enable ports based on the pipeline selections.
    - **Cascadable Ports:**
        - **Provide pcin port:** When selected, the `pcin` port is exposed. The `pcin` port must be connected to the `pcout` port of another DSP48 block.
        - **Provide carry cascade in port:** When selected, the carry cascade in port is exposed. This port can only be connected to a carry cascade out port on another DSP48E block.
        - **Provide multiplier sign cascade in port:** When selected, the multiplier sign cascade in port (multsigncascin) is exposed. This port can only be connected to a multiplier sign cascade out port of another DSP48E block.
    - **Output Ports:**
        - **Provide carryout port:** When selected, the `carryout` output port is made available. When the mode of operation for the adder/subtractor is set to one 48-bit adder, the `carryout` port is 1-bit wide. When the mode of operation is set to two 24 bit adders, the `carryout` port is 2 bits wide. The MSB corresponds to the second adder's carryout and the LSB corresponds to the first adder's carryout. When the mode of operation is set to four 12 bit adders, the `carryout` port is 4 bits wide with the bits corresponding to the addition of the 48 bit input split into 4 12-bit sections.
        - **Provide pattern detect port:** When selected, the pattern detection output port is provided. When the pattern, either from the mask or the c register, is matched the pattern detection port is set to '1'.

  - **Provide pattern bar detect port:** When selected, the pattern bar detection (patternbdetect) output port is provided. When the inverse of the pattern, either from the mask or the c register, is matched the pattern bar detection port is set to '1'.
  - **Provide overflow port:** When selected, the overflow output port is provided. This port indicates when the operation in the DSP48E has overflowed beyond the bit P[N] where N is between 1 and 46. N is determined by the number of 1s in the mask whether set by the GUI mask field or the c port input.
  - **Provide underflow port:** When selected, the underflow output port is provided. This port indicates when the operation in the DSP48E has underflowed. Underflow occurs when the number goes below –P[N] where N is determined by the number of 1s in the mask whether set by the GUI mask field or the c port input.
- **Cascadable Ports:**
  - **Provide acout port:** When selected, the `acout` output port is made available. The `acout` port must be connected to the `acin` port of another DSP48E block.
  - **Provide bcout port:** When selected, the `bcout` output port is made available. The `bcout` port must be connected to the `bcin` port of another DSP48E block.
  - **Provide pcout port:** When selected, the `pcout` output port is made available. The `pcout` port must be connected to the `pcin` port of another DSP48 block.
  - **Provide multiplier sign cascade out port:** When selected, the multiplier sign cascade out port (multsigncascout) is made available. This port can only be connected to the multiplier sign cascade in port of another DSP48E block and is used to support 96-bit accumulators/adders and subtracters which are built from two DSP48Es.
  - **Provide carry cascade out port:** When selected, the carry cascade out port (`carrycascout`) is made available. This port can only be connected to the carry cascade in port of another DSP48E block.

- **Pipelining tab:**

  Parameters specific to the Pipelining tab are as follows.

  - **Length of a/acin pipeline:** Specifies the length of the pipeline on input register A. A pipeline of length 0 removes the register on the input.
  - **Length of b/bcin pipeline:** Specifies the length of the pipeline for the b input whether it is read from b or `bcin`.
  - **Length of acout pipeline:** Specifies the length of the pipeline between the a/`acin` input and the `acout` output port. A pipeline of length 0 removes the register from the `acout` pipeline length. Must be less than or equal to the length of the a/`acin` pipeline.
  - **Length of bcout pipeline:** Specifies the length of the pipeline between the b/`bcin` input and the bcout output port. A pipeline of length 0 removes the register from the bcout pipeline length. Must be less than or equal to the length of the b/`bcin` pipeline.

- **Pipeline c:** Indicates whether the input from the c port should be registered.
- **Pipeline p:** Indicates whether the outputs p and pcout should be registered.
- **Pipeline multiplier:** Indicates whether the internal multiplier should register its output.
- **Pipeline opmode:** Indicates whether the opmode port should be registered.
- **Pipeline alumode:** Indicates whether the alumode port should be registered.
- **Pipeline carry in:** Indicates whether the carry in port should be registered.
- **Pipeline carry in select:** Indicates whether the carry in select port should be registered.
- **Pipeline preadder input register d:** Indicates to add a pipeline register to the d input.
- **Pipeline preadder output register ad:** Indicates to add a pipeline register to the ad output.
- **Pipeline INMODE register:** Indicates to add a pipeline register to the INMODE input.

- **Reset/Enable Ports tab:** Parameters specific to the Reset/Enable tab are as follows.
  - **Provide Reset Ports:**
    - **Reset port for a/acin:** When selected, a port rst_a is made available. This resets the pipeline register for port a when set to '1'.
    - **Reset port for b/bcin:** When selected, a port rst_b is made available. This resets the pipeline register for port b when set to '1'.
    - **Reset port for c:** When selected, a port rst_c is made available. This resets the pipeline register for port c when set to '1'.
    - **Reset port for multiplier:** When selected, a port rst_m is made available. This resets the pipeline register for the internal multiplier when set to '1'.
    - **Reset port for P:** When selected, a port rst_p is made available. This resets the output register when set to '1'.
    - **Reset port for carry in:** When selected, a port rst_carryin is made available. This resets the pipeline register for carry in when set to '1'.
    - **Reset port for alumode:** When selected, a port rst_alumode is made available. This resets the pipeline register for the alumode port when set to '1'.
    - **Reset port for controls (opmode and carry_in_sel):** When selected, a port rst_ctrl is made available. This resets the pipeline register for the opmode register (if available) and the carry_in_sel register (if available) when set to '1'.
    - **Reset port for d and ad:** When selected, a port rst_a and rst_ad is made available. This resets the pipeline register for ports when set to '1'.

- **Reset port for INMODE:** When selected, a port rst_inmode is made available. This resets the pipeline register for the inmode port when set to '1'.

- **Provide Enable Ports:**
  - **Enable port for first a/acin register:** When selected, an enable port ce_a1 for the first a pipeline register is made available.
  - **Enable port for second a/acin register:** When selected, an enable port ce_a2 for the second a pipeline register is made available.
  - **Enable port for first b/bcin register:** When selected, an enable port ce_b1 for the first b pipeline register is made available.
  - **Enable port for second b/bcin register:** When selected, an enable port ce_b2 for the second b pipeline register is made available.
  - **Enable port for c:** When selected, an enable port ce_c for the port C register is made available.
  - **Enable port for multiplier:** When selected, an enable port ce_m for the multiplier register is made available.
  - **Enable port for p:** When selected, an enable port ce_p for the port P output register is made available.
  - **Enable port for carry in:** When selected, an enable port ce_carry_in for the carry in register is made available.
  - **Enable port for alumode:** When selected, an enable port ce_alumode for the alumode register is made available.
  - **Enable port for multiplier carry in:** When selected, an enable port mult_carry_in for the multiplier register is made available.
  - **Enable port for controls (opmode and carry_in_sel):** When selected, the enable port ce_ctrl is made available. The port ce_ctrl controls the opmode and carry in select registers.
  - **Enable port for d:** When selected, an enable port is added input register d.
  - **Enable port for ad:** When selected, an enable port is add for the preadder output register ad.
  - **Enable port for INMODE:** When selected, an enable port is added for the INMODE register.

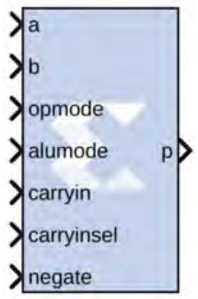
- **Inversion Options tab:** When a checkbox is selected on this tab, the specified signal is inverted.
- **Implementation tab:** Parameters specific to the Implementation tab are as follows.

- **Use synthesizable model:** When selected, the DSP48E2 is implemented from an RTL description which might not map directly to the DSP48E2 hardware. This is useful if a design using the DSP48E2 block is targeted at device families that do not contain DSP48E2 hardware primitives.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

DSP58

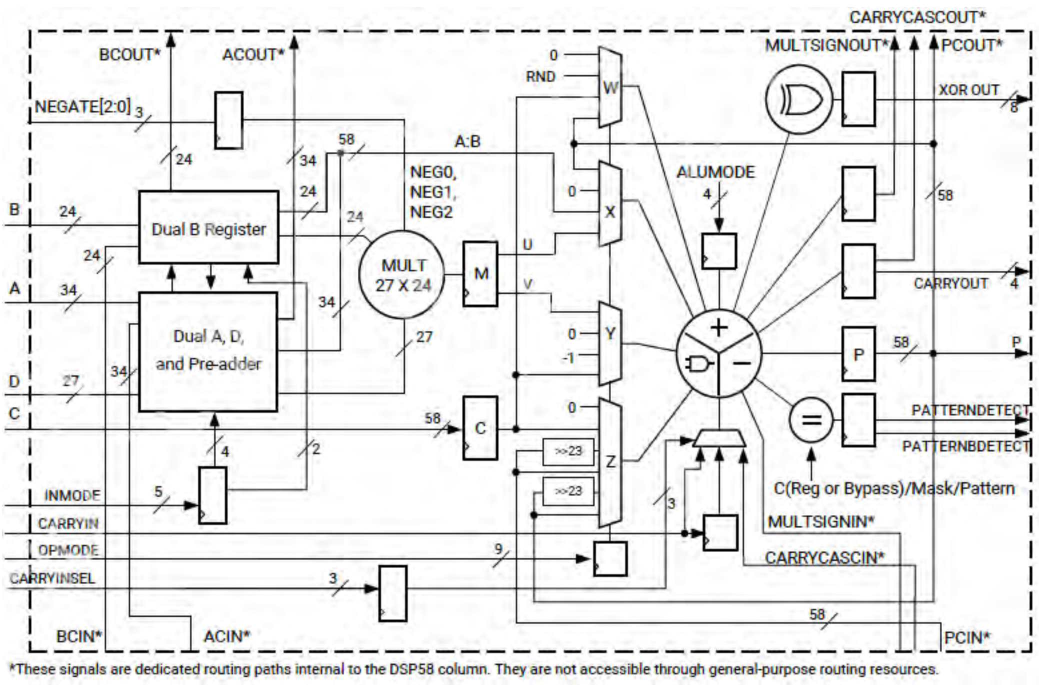
The Xilinx DSP58 block is an efficient building block for DSP applications that use Versal™ devices. DSP applications use many binary multipliers and accumulators that are best implemented in dedicated DSP resources. Versal™ devices have many dedicated low-power DSP slices, combining high speed with small size while retaining system design flexibility.



The DSP58 slice is effectively a super-set of the DSP48E2 slice with these differences. The DSP58 has the following.

- Wider functionality
- More flexibility in the pre-adder
- New optional negate inport
- More XOR operations
- Additional unique features

Figure 327: **DSP58**



*These signals are dedicated routing paths internal to the DSP58 column. They are not accessible through general-purpose routing resources.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**
  - **Input Configuration:**
    - **A or ACIN input:** Specifies if the A input should be taken directly from the a port or from the cascaded acin port. The acin port can only be connected to another DSP58 block.
    - **B or BCIN input:** Specifies if the B input should be taken directly from the b port or from the cascaded bcin port. The bcin port can only be connected to another DSP58 block.

    Note: If the input to the block is NaN, you will see a behavioral simulation mismatch.
  - **DSP58 Data-Path Configuration:**
    - **SIMD Mode of Adder/Subtractor/Accumulator:** This mode can be used to implement small add-subtract functions at high speed and lower power with less logic utilization. The adder and subtracter in the adder/subtracted/logic unit can also be split into two 24-bit units or four 12-bit units.
    - **Mode of Multiplier :** This option is disabled in the current release

- **Do not use multiplier:** When this is selected, the DSP58 block is optimized in hardware for maximum performance without using the multiplier. If an instruction using the multiplier is encountered in simulation, an error is reported.
- **Use dynamic multiplier mode:** When this is selected, it instructs the DSP58 block to use the dynamic multiplier mode. This indicates that the block is switching between A*B and A:B operations on the fly, and therefore needs to get the worst-case timing of the two paths.

- **Preadder Configuration:** Use the 27-bit D data input to the pre-adder or alternative input to the multiplier. The pre-adder implements D + A as determined by the INMODE3 signal.
  - **PREADDINSEL Select preadder input:** Selects the input to be added with D in the pre-adder.
  - **AMULTSEL Select A multiplexer output:** Selects the input to the 27-bit A input of the multiplier. In the 7 series primitive, DSP48E1 the attribute is called USE_DPORT, but has been renamed due to new pre-adder flexibility enhancements (default AMULTSEL = A is equivalent to USE_DPORT=FALSE).
  - **BMULTSEL Select B multiplexer output:** Selects the input to the 18-bit B input of the multiplier.
  - **Enable D Port:** Automatically enabled when AD is selected.
- **Pattern Detection:**
  - **Reset p register on pattern detection:** If selected and the pattern is detected, reset the p register on the next cycle
  - **AUTO RESET PRIORITY:** When enabled by selecting the option above, select RESET (the default) or CEP (clock enabled for the P (output) resister).
- **Pattern Input:**
  - **Pattern Input from c port:** When selected, the pattern used in pattern detection is read from the c port.
  - **Using Pattern Attribute (58bit hex value):** Value is used in pattern detection logic, which is best described as an equality check on the output of the adder/subtractor/logic unit.
  - **Pattern attribute:** A 58-bit value that is used in the pattern detector.
- **Mask Input:**
  - **Mask input from c port:** When selected, the mask used in pattern detection is read from the c port.
  - **Using Mask Attribute (58 bit hex value):** A 58-bit value used to mask out certain bits during pattern detection.
  - **MODE1:** Selects rounding_mode 1 (C-bar left shifted by 1).

    - **MODE2:** Selects rounding_mode 2 (C-bar left shifted by 2).

- **Wide Xor tab:** Parameters specific to the Wide Xor tab are as follows.

  - **Use Wide XOR:** Use this is feature to perfom a 116 bit XOR function.

  - **XORSIMD Select Wide XOR SIMD:** Use the XORSIMD attribute to select the width of the XOR function. Select either XOR12 (the default), XOR22, XOR24, XOR34, XOR58, or XOR116).

- **Optional Ports tab:**

  - **Input Ports:**

    - **Consolidate control port:** When selected, combines the opmode, alumode, carry_in, carry_in_sel, inmode, and negate ports into one 25-bit port. Bits 0 to 8 are the opmode, bits 9 to 12 are the alumode port, bit 13 is the carry_in port, and bits 14 to 16 are the carry_in_sel port, bits 17 to 21 are the inmode port, and bits 22 to 24 are the negate port. This option should be used when the Opmode block is used to generate a DSP58 instruction.

    - **Provide c port:** When selected, the c port is made available. Otherwise, the c port is tied to '0'.

    - **Provide global reset port:** When selected, the port rst is made available. This port is connected to all available reset ports based on the pipeline selections.

    - **Provide global enable port:** When selected, the optional en port is made available. This port is connected to all available enable ports based on the pipeline selections.

  - **Cascadable Ports:**

    - **Provide pcin port:** When selected, the pcin port is exposed. The pcin port must be connected to the pcout port of another DSP58 block.

    - **Provide carry cascade in port:** When selected, the carry cascade in port is exposed. This port can only be connected to a carry cascade out port on another DSP58 block.

    - **Provide multiplier sign cascade in port:** When selected, the multiplier sign cascade in port (multsigncascin) is exposed. This port can only be connected to a multiplier sign cascade out port of another DSP58 block.

  - **Output Ports:**

    - **Provide carryout port:** When selected, the carryout output port is made available. When the mode of operation for the adder/subtractor is set to one 58-bit adder, the carryout port is 1-bit wide. When the mode of operation is set to two 24 bit adders, the carryout port is 2 bits wide. The MSB corresponds to the second adder's carryout and the LSB corresponds to the first adder's carryout. When the mode of operation is set to four 12 bit adders, the carryout port is 4 bits wide with the bits corresponding to the addition of the 48 bit input split into four 12-bit sections.

- **Provide pattern detect port:** When selected, the pattern detection output port is provided. When the pattern, either from the mask or the c register, is matched the pattern detection port is set to '1'.
- **Provide pattern bar detect port:** When selected, the pattern bar detection (patternbdetect) output port is provided. When the inverse of the pattern, either from the mask or the c register, is matched the pattern bar detection port is set to '1'.
- **Provide overflow port:** When selected, the overflow output port is provided. This port indicates when the operation in the DSP58 has overflowed beyond the bit P[N] where N is between 1 and 46. N is determined by the number of 1s in the mask whether set by the GUI mask field or the c port input.
- **Provide underflow port:** When selected, the underflow output port is provided. This port indicates when the operation in the DSP58 has underflowed. Underflow occurs when the number goes below –P[N] where N is determined by the number of 1s in the mask whether set by the GUI mask field or the c port input.

- **Cascadable Ports:**
  - **Provide ACOUT port:** When selected, the acout output port is made available. The acout port must be connected to the acin port of another DSP58 block.
  - **Provide BCOUT port:** When selected, the bcout output port is made available. The bcout port must be connected to the bcin port of another DSP58 block.
  - **Provide PCOUT port:** when selected, the pcout output port is made available. The pcout port must be connected to the pcin port of another DSP58 block.
  - **Provide multiplier sign cascade out port:** When selected, the multiplier sign cascade out port (multsigncascout) is made available. This port can only be connected to the multiplier sign cascade in port of another DSP58 block and is used to support 96-bit accumulators/adders and subtracters which are built from two DSP58s.
  - **Provide carry cascade out port:** When selected, the carry cascade out port (carrycascout) is made available. This port can only be connected to the carry cascade in port of another 58 block.

- **Pipelining tab:**
  - **Pipeline Options:**
    - **Length of a/acin pipeline:** Specifies the length of the pipeline on input register A. A pipeline of length 0 removes the register on the input.
    - **Length of b/bCIN pipeline:** Specifies the length of the pipeline for the b input whether it is read from b or bcin.
    - **Length of acout pipeline:** Specifies the length of the pipeline between the a/acin input and the acout output port. A pipeline of length 0 removes the register from the acout pipeline length. Must be less than or equal to the length of the a/acin pipeline.

        - **Length of bcout pipeline:** Specifies the length of the pipeline between the b/bcin input and the bcout output port. A pipeline of length 0 removes the register from the bcout pipeline length. Must be less than or equal to the length of the b/bcin pipeline.
        - **Pipeline c:** Indicates whether the input from the c port should be registered.
        - **Pipeline p:** Indicates whether the outputs p and pcout should be registered.
        - **Pipeline multiplier:** indicates whether the internal multiplier should register its output.
        - **Pipeline opmode:** Indicates whether the opmode port should be registered.
        - **Pipeline alumode:** Indicates whether the alumode port should be registered.
        - **Pipeline carry in:** Indicates whether the carry in port should be registered.
        - **Pipeline carry in select:** Indicates whether the carry in select port should be registered.
        - **Pipeline preadder input register d:** Indicates to add a pipeline register to the d input.
        - **Pipeline preadder output register ad:** Indicates to add a pipeline register to the ad output.
        - **Pipeline INMODE register:** Indicates to add a pipeline register to the INMODE input.

- **Reset/Enable Ports tab:**
    - **Provide Reset Ports:**
        - **Reset port for a/acin:** When selected, a port rst_a is made available. This resets the pipeline register for port a when set to '1'.
        - **Reset port for b/bcin:** When selected, a port rst_b is made available. This resets the pipeline register for port b when set to '1'.
        - **Reset port for c:** When selected, a port rst_c is made available. This resets the pipeline register for port c when set to '1'.
        - **Reset port for multiplier:** when selected, a port rst_m is made available. This resets the pipeline register for the internal multiplier when set to '1'.
        - **Reset port for P:** When selected, a port rst_p is made available. This resets the output register when set to '1'.
        - **Reset port for carry in:** When selected, a port rst_carryin is made available. This resets the pipeline register for carry in when set to '1'.
        - **Reset port for alumode:** When selected, a port rst_alumode is made available. This resets the pipeline register for the alumode port when set to '1'.
        - **Reset port for controls (opmode and carry_in_sel):** When selected, a port rst_ctrl is made available. This resets the pipeline register for the opmode register (if available) and the carry_in_sel register (if available) when set to '1'.

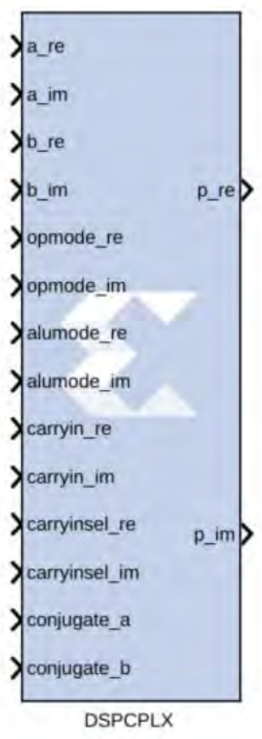
- **Provide Enable Ports:**
    - **Enable port for first a/acin register:** When selected, an enable port ce_a1 for the first a pipeline register is made available.
    - **Enable port for second a/acin register:** When selected, an enable port ce_a2 for the second a pipeline register is made available.
    - **Enable port for first b/bcin register:** When selected, an enable port ce_b1 for the first b pipeline register is made available.
    - **Enable port for second b/bcin register:** When selected, an enable port ce_b2 for the second b pipeline register is made available.
    - **Enable port for c:** When selected, an enable port ce_c for the port C register is made available.
    - **Enable port for multiplier:** When selected, an enable port ce_m for the multiplier register is made available.
    - **Enable port for p:** When selected, an enable port ce_p for the port P output register is made available.
    - **Enable port for carry in:** When selected, an enable port ce_carry_in for the carry in register is made available.
    - **Enable port for alumode:** When selected, an enable port ce_alumode for the alumode register is made available.
    - **Enable port for multiplier carry in:** When selected, an enable port mult_carry_in for the multiplier register is made available.
    - **Enable port for controls (opmode and carry_in_sel):** When selected, the enable port ce_ctrl is made available. The port ce_ctrl controls the opmode and carry in select registers.
    - **Enable port for d:** When selected, an enable port is added input register d.
    - **Enable port for ad:** When selected, an enable port is add for the preadder output register ad.
    - **Enable port for INMODE:** When selected, an enable port is added for the INMODE register.
- **Inversion Options tab :** When a checkbox is selected under this tab, the specified signal is inverted.
- **Implementation tab:**

- **Use synthesizable model:** When selected, the DSP58 is implemented from an RTL description which might not map directly to the DSP58 hardware. This is useful if a design using the DSP58 block is targeted at device families that do not contain DSP58 hardware primitives.
- **Mode of operation for the adder/subtractor:** This mode can be used to implement small add-subtract functions at high speed and lower power with less logic utilization. The adder and subtractor in the adder/subtracted/logic unit can also be split into two 24-bit fields or four12-bit fields. This is achieved by setting the mode of operation to "Two 24-bit adders" or "Four 12-bit adders".

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

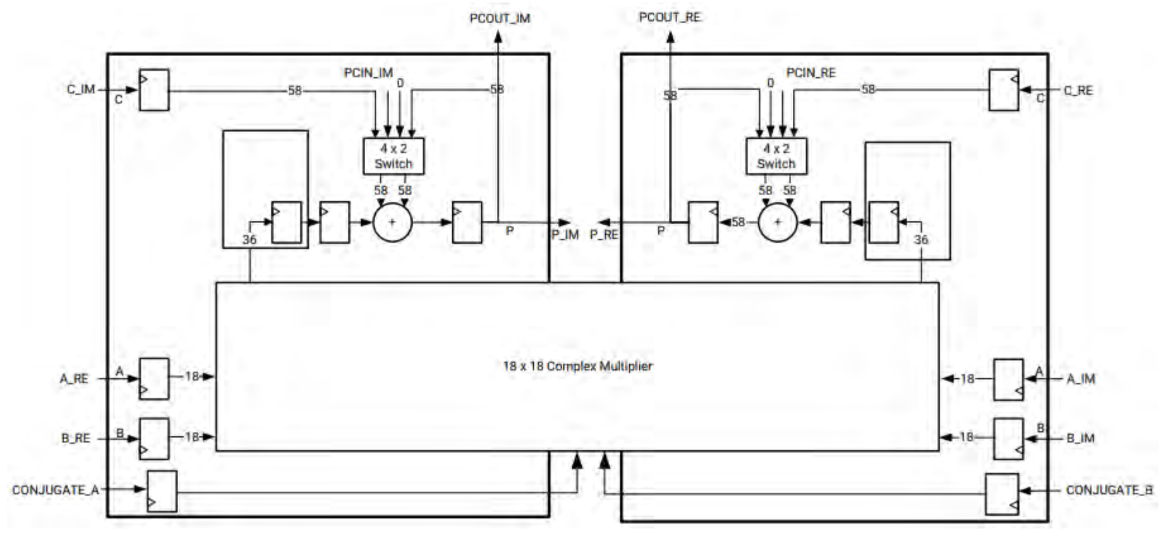
DSPCPLX

The Xilinx DSPCPLX block is one of the advanced features provided by Versal™ architecture DSP, which is the optimized solution to deal with 18x18 complex multiplication followed by 58 + 58 accumulation operation.



Versal architecture DSP supports an 18-bit complex multiplier with two back-to-back DSP58s in the same tile pair together. The two DSP58s with their DSP_MODE attributes set to CINT18 form one complex arithmetic unit. The right DSP58 computes the real result P_RE and left computes the imaginary result P_IM. The following figure shows the unisim DSPCPLX primitive which is used to develop this feature.

Figure 328: **Unisim DSPCPLX Primitive**



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

- **Basic tab:** Parameters specific to the Basic tab are as follows:
  - **Input Configuration:**
    - **A or ACIN input:** Specifies if the A input should be taken directly from the `a_re`, `a_im` ports or from the cascaded `acin_re`, `acin_im` ports. The `acin_re` and `acin_im` ports can only be connected to another DSPCPLX block.
    - **B or BCIN input:** Specifies if the B input should be taken directly from the b_re, b_im ports or from the cascaded bcin_re, bcin_im ports. The bcin_re and bcin_im ports can only be connected to another DSPCPLX block.
  - **Pattern Detection on Real Output:**
    - **Reset p_re register on pattern detection:** If selected and the `pattern_re` is detected, reset the `p_re` register on the next cycle
    - **AUTO RESET PRIORITY RE:** When enabled by selecting the option above, select RESET (the default) or CEP (clock enabled for the P_RE (output) resister).

- **Pattern Input RE:**
  - **Pattern Input from c_re port:** When selected, the `pattern_re` used in pattern detection on Real Output is read from the `c_re` port.
  - **Using Pattern Attribute RE (58-bit hex value):** Value is used in pattern detection logic which is best described as an equality check on the output of the adder/subtractor/logic unit.
  - **Using Pattern Attribute RE (58-bit hex value):** Enter a 58-bit value that is used in the pattern detector.
- **Mask Input RE:**
  - **Mask input from c_re port:** When selected, the `mask_re` used in pattern detection is read from the `c_re` port.
  - **Using Mask Attribute RE (58-bit hex value):** Enter a 58-bit value used to mask out certain bits during pattern detection on Real Output.
  - **MODE1:** Selects rounding_mode 1 (C_RE-bar left shifted by 1).
  - **MODE2:** Selects rounding_mode 2 (C_RE-bar left shifted by 2).
- **Pattern Detection on Imaginary Output:**
  - **Reset p_im register on pattern detection:** If selected and the pattern_im is detected, reset the p_im register on the next cycle.
  - **AUTO RESET PRIORITY IM:** When enabled by selecting the option above, select RESET (the default) or CEP (clock enabled for the P_IM (output) resister).
- **Mask Input IM:**
  - **Mask input from c_im port:** When selected, the `mask_im` used in pattern detection on Imaginary Output is read from the `c_im` port.
  - **Using Mask Attribute RE (58-bit hex value):** Enter a 58-bit value used to mask out certain bits during pattern detection on Imaginary Output.
  - **MODE1:** Selects rounding_mode 1 (C_IM-bar left shifted by 1).
  - **MODE2:** Selects rounding_mode 2 (C_IM-bar left shifted by 2).

- **Optional Ports tab:** Parameters specific to the Optional Ports tab are as follows:
  - **Input Ports:**

- **Consolidate control port:** When selected, combines the opmode, alumode, carry_in, carry_in_sel, inmode and conjugate ports into one 18-bit port. Bits 0 to 8 are the opmode, bits 9 to 12 are the alumode port, bit 13 is the carry_in port, bits 14 to 16 are the carry_in_sel port, bit 17 is the Conjugate_A input port and bit 18 is the Conjugate_B port. This option should be used when the Opmode block is used to generate a DSPCPLX instruction.

  Note: Enabling this option will drive both the left and right dsp58 tiles with the same configuration.

- **Provide c port:** When selected, the c_re and c_im ports are made available. Otherwise, the c_re and c_im ports are tied to '0'.
- **Provide global reset port:** When selected, the port rst_all is made available. This port is connected to all available reset ports based on the pipeline selections.
- **Provide global enable port:** When selected, the optional en_all port is made available. This port is connected to all available enable ports based on the pipeline selections.

- **Cascadable Ports:**
  - **Provide pcin port:** When selected, the pcin_re and pcin_im ports are exposed. The pcin_re and pcin_im ports must be connected to the pcout_re and pcout_im ports of another DSPCPLX block respectively.
  - **Provide carry cascade in port:** When selected, the carrycascin_re and carrycascin_im ports are exposed. These ports can only be connected to a carry cascade out ports of another DSPCPLX block.
  - **Provide multiplier sign cascade in port:** When selected, the multsignin_re and multsignin_im ports are exposed. These ports can only be connected to a multiplier sign cascade out ports of another DSPCPLX block.
- **Output Ports:**
  - **Provide carryout port:** When selected, the carryout_re and carryout_im output ports are made available.
  - **Provide pattern detect port:** When selected, the patterndetect_re and patterndetect_out ports are provided. When the pattern_re/pattern_im, either from the mask_re/mask_im or the c_re/c_im register is matched, the respective patterndetect_re/patterndetect_im port is set to '1'.
  - **Provide pattern bar detect port:** When selected, the patternbdetect_re and patternbdetect_im ports are provided. When the inverse of the pattern_re/ pattern_im, either from the mask_re/mask_im or the c_re/c_im register is matched, the patternbdetect_re/patternbdetect_im port is set to '1'.

- **Provide overflow port:** When selected, the `overflow_re` and `overflow_im` ports are provided. These ports indicate when the operation in the DSPCPLX has overflowed beyond the bit P_RE[N]/P_IM[N] where N is between 0 and 56. N is determined by the number of 1s in the `mask_re/mask_im` whether set by the GUI mask field or the `c_re/c_im` port input.
- **Provide underflow port:** When selected, the `underflow_re` and `underflow_im` ports are provided. These ports indicate when the operation in the DSPCPLX has underflowed. Underflow occurs when the number goes below -P_RE[N]/P_IM[N], where N is determined by the number of 1s in the `mask_re/mask_im` whether set by the GUI mask field or the `c_re/c_im` port input.

- **Cascadable Ports:**
  - **Provide acout port:** When selected, the `acout_re` and `acout_im` output ports are made available. The `acout_re/acout_im` port must be connected to the `acin_re/acin_im` port of another DSPCPLX block.
  - **Provide bcout port:** When selected, the `bcout_re` and `bcout_im` output ports are made available. The `bcout_re/bcout_im` port must be connected to the `bcin_re/bcin_im` port of another DSPCPLX block.
  - **Provide pcout port:** When selected, the `pcout_re` and `pcout_im` output ports are made available. The `pcout_re/pcout_im` port must be connected to the `pcin_re/pcin_im` port of another DSPCPLX block.
  - **Provide multiplier sign cascade out port:** When selected, the `multsignout_re` and `multsignout_im` ports are made available. These ports can only be connected to the `multsignin_re` and `multsignin_im` ports of another DSPCPLX block respectively and is used to support 116-bit accumulators/adders and subtracters which are built from two DSPCPLXs.
  - **Provide carry cascade out port:** When selected, the `carrycascout_re` and `carrycascout_im` ports are made available. These ports can only be connected to the `carrycascin_re` and `carrycascin_im` ports of another DSPCPLX block respectively.

- **Pipelining tab:** Parameters specific to the Pipelining tab are as follows:
  - **Length of a_re/acin_re pipeline:** Specifies the length of the pipeline on input register A_RE. The pipeline of length 0 removes the register on the input.
  - **Length of a_im/acin_im pipeline:** Specifies the length of the pipeline on input register A_IM. The pipeline of length 0 removes the register on the input.
  - **Length of b_re/bcin_re pipeline:** Specifies the length of the pipeline for the `b_re` input and whether it is read from `b_re` or `bcin_re`.
  - **Length of b_im/bcin_im pipeline:** Specifies the length of the pipeline for the `b_im` input and whether it is read from `b_im` or `bcin_im`.

- **Length of acout_re pipeline:** Specifies the length of the pipeline between the `a_re/acin_re` input and the `acout_re` output port. The pipeline of length 0 removes the register from the `acout_re` pipeline length. Must be less than or equal to the length of the `a_re/acin_re` pipeline.
- **Length of acout_im pipeline:** Specifies the length of the pipeline between the `a_im/acin_im` input and the `acout_im` output port. The pipeline of length 0 removes the register from the `acout_im` pipeline length. Must be less than or equal to the length of the `a_im/acin_im` pipeline.
- **Length of bcout_re pipeline:** Specifies the length of the pipeline between the `b_re/bcin_re` input and the `bcout_re` output port. The pipeline of length 0 removes the register from the `bcout_re` pipeline length. Must be less than or equal to the length of the `b_re/bcin_re` pipeline.
- **Length of bcout_im pipeline:** Specifies the length of the pipeline between the `b_im/bcin_im` input and the `bcout_im` output port. The pipeline of length 0 removes the register from the `bcout_im` pipeline length. Must be less than or equal to the length of the `b_im/bcin_im` pipeline.
- **Pipeline c_re:** Indicates whether the input from the `c_re` port should be registered.
- **Pipeline c_im:** Indicates whether the input from the `c_im` port should be registered.
- **Pipeline p_re:** Indicates whether the outputs `p_re` and `pcout_re` should be registered.
- **Pipeline p_im:** Indicates whether the outputs `p_im` and `pcout_im` should be registered.
- **Pipeline multiplier_re:** Indicates whether the internal `multiplier_re` should register its output.
- **Pipeline multiplier_im:** Indicates whether the internal `multiplier_im` should register its output.
- **Pipeline opmode_re:** Indicates whether the `opmode_re` port should be registered.
- **Pipeline opmode_im:** Indicates whether the `opmode_im` port should be registered.
- **Pipeline alumode_re:** Indicates whether the `alumode_re` port should be registered.
- **Pipeline alumode_im:** Indicates whether the `alumode_im` port should be registered.
- **Pipeline carry in Re:** Indicates whether the `carryin_re` port should be registered.
- **Pipeline carry in Im:** Indicates whether the `carryin_im` port should be registered.
- **Pipeline carry in select Re:** Indicates whether the `carryinsel_re` port should be registered.
- **Pipeline carry in select Im:** Indicates whether the `carryinsel_im` port should be registered.

- **Pipeline preadder output register ad:** Indicates to add a pipeline register to the `ad` output.
- **Pipeline Conjugate register A:** Indicates to add a pipeline register to the `Conjugate_A` input.
- **Pipeline Conjugate register B:** Indicates to add a pipeline register to the `Conjugate_B` input.

- **Reset/Enable Ports tab:** Parameters specific to the Reset/Enable tab are as follows:
    - **Provide Reset Ports:**
        - **Reset port for a/acin:**

          When selected, `rsta_re` and `rsta_im` ports are made available. This resets the pipeline registers for port `a_re`, `a_im` when set to '1'.
        - **Reset port for b/bcin:** When selected, `rstb_re` and `rstb_im` are made available. This resets the pipeline registers for port `b_re`, `b_im` when set to '1'.
        - **Reset port for c:** When selected, `rstc_re` and `rstc_im` are made available. This resets the pipeline registers for port `c_re`, `c_im` when set to '1'.
        - **Reset port for multiplier:** When selected, `rstm_re` and `rstm_im` are made available. This resets the pipeline registers for internal multiplier respectively when set to '1'.
        - **Reset port for P:** When selected, `rstp_re` and `rstp_im` are made available. This resets the output `p_re` and `p_im` registers when set to '1'.
        - **Reset port for carry in:** When selected, `rstallcarryin_re` and `rstallcarryin_im` are made available. This resets the pipeline registers for `carryin_re` and `carryin_im` port when set to '1'.
        - **Reset port for alumode:** When selected, `rstalumode_re` and `rstalumode_im` are made available. This resets the pipeline register for the `alumode_re` and `alumode_im` port when set to '1'.
        - **Reset port for controls (opmode and carry_in_sel):** When selected, a port `rstctrl_re` and `rstctrl_im` are made available. This resets the pipeline register for the `opmode_re/opmode_im` register (if available) and the `carryinsel_re/ carryinsel_im` register (if available) when set to '1'.
        - **Reset port for ad:** When selected, port `rstad` is made available. This resets the pipeline ad register for ports when set to '1'.
        - **Reset port for Conjugate_a:** When selected, port `rstconjugate_a` is made available. This resets the pipeline register for the `Conjugate_a` port when set to '1'.
        - **Reset port for Conjugate_b:** When selected, port `rstconjugate_b` is made available. This resets the pipeline register for the `Conjugate_b` port when set to '1'.

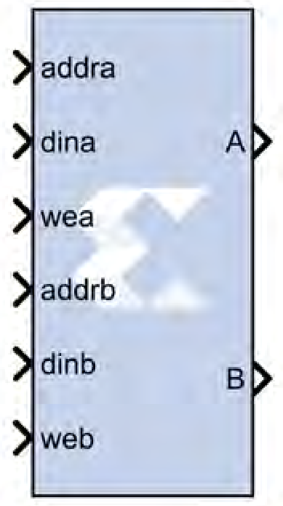
- **Provide Enable Ports:**
  - **Enable port for first a/acin register:** When selected, enable ports `cea1_re` and `cea1_im` for the first `a_re` and `a_im` pipeline register are made available.
  - **Enable port for second a/acin register:** When selected, enable ports `cea2_re` and `cea2_im` for the second `a_re` and `a_im` pipeline registers are made available.
  - **Enable port for first b/bcin register:** When selected, enable ports `ceb1_re` and `ceb1_im` for the first `b_re` and `b_im` pipeline registers are made available.
  - **Enable port for second b/bcin register:** When selected, enable ports `ceb2_re` and `ceb2_im` for the second `b_re` and `b_im` pipeline registers are made available.
  - **Enable port for c:** When selected, enable ports `cec_re` and `cec_im` for the port `C_re` and `C_im` registers are made available.
  - **Enable port for multiplier:** When selected, enable ports `cem_re` and `cem_im` for the Real and Imaginary multiplier registers are made available.
  - **Enable port for p:** When selected, enable ports `cep_re` and `cep_im` for the port `P_re` and `P_im` output registers are made available.
  - **Enable port for carry in:** When selected, enable ports `cecarryin_re` and `cecarryin_im` for the Real and Imaginary carry in registers are made available.
  - **Enable port for alumode:** When selected, enable ports `cealumode_re` and `cealumode_im` for the Real and Imaginary alumode registers are made available.
  - **Enable port for controls (opmode and carry_in_sel):** When selected, enable ports `cectrl_re` and `cectrl_im` are made available. The ports `cectrl_re` and `cectrl_im` controls the Real and Imaginary opmode and carry in select registers.
  - **Enable port for ad:** When selected, an `enable` port is created for the preadder output register ad.
  - **Enable port for Conjugate_a:** When selected, an enable port `conjugate_a` is added for the `Conjugate_A` register.
  - **Enable port for Conjugate_b:** When selected, an enable port `conjugate_b` is added for the `Conjugate_B` register.

- **Inversion Options tab:** When the checkbox is selected on this tab, the specified signal is inverted.
- **Implementation tab:** Parameters specific to the Implementation tab are as follows.
  - **Use synthesizable model:** When selected, the DSPCPLX is implemented from an RTL description which might not map directly to the DSP58 hardware. This is useful if a design using the DSPCPLX block is targeted at device families that do not contain DSP58 hardware primitives.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Dual Port RAM

The Xilinx Dual Port RAM block implements a random access memory (RAM). Dual ports enable simultaneous access to the memory space at different sample rates using multiple data widths.



Block Interface

The block has two independent sets of ports for simultaneous reading and writing. Independent address, data, and write enable ports allow shared access to a single memory space. By default, each port set has one output port and three input ports for address, input data, and write enable. Optionally, you can also add a port enable and synchronous reset signal to each input port set.

A dual-port RAM can be implemented using either distributed memory, block RAM, or UltraRAM resources in the FPGA.

Form Factors

The Dual Port RAM block also supports various Form Factors (FF). Form factor is defined as:

$$FF = W_B / W_A$$

where W_B is data width of Port B and W_A is Data Width of Port A.

The Depth of port B (D_B) is inferred from the specified form factor as follows:

$$D_B = D_A / FF$$

The data input ports on Port A and B can have different arithmetic type and binary point position for a form factor of 1. For form factors greater than 1, the data input ports on Port A and Port B should have an unsigned arithmetic type with binary point at 0. The output ports, labeled A and B, have the same types as the corresponding input data ports.

The location in the memory block can be accessed for reading or writing by providing the valid address on each individual address port. A valid address is an unsigned integer from 0 to d-1, where d denotes the RAM depth (number of words in the RAM) for the particular port. An attempt to read past the end of the memory is caught as an error in simulation. When the dual-port RAM is implemented in distributed memory or block RAM, the initial RAM contents can be specified through a block parameter. Each write enable port must be a boolean value. When the WE port is 1, the value on the data input is written to the location indicated by the address line.

Write Mode

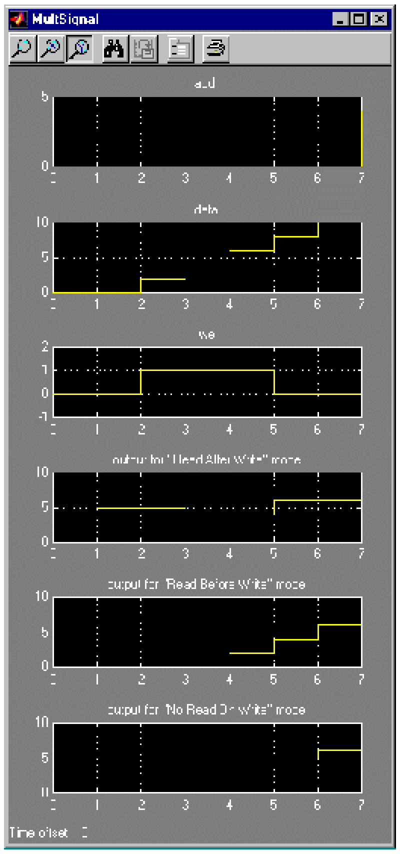
When the Dual Port RAM block is implemented in block RAM, you can set the write mode for the block in the block parameters dialog box.

The output during a write operation depends on the write mode. When the WE is 0, the output port has the value at the location specified by the address line. During a write operation (WE asserted), the data presented on the input data port is stored in memory at the location selected by the port's address input. During a write cycle, you can configure the behavior of each data out port A and B to one of the following choices:

- **Read after write**
- **Read before write**
- **No read on write**

The write modes can be described with the help of the figure below. In the figure, the memory has been set to an initial value of 5 and the address bit is specified as 4. When using **No read on write** mode, the output is unaffected by the address line and the output is the same as the last output when the WE was 0. For the other two modes, the output is obtained from the location specified by the address line, and hence is the value of the location being written to. This means that the output can be the old value which corresponds to **Read after write**.

Figure 329: **Write Mode Output**



Collision Behavior

The result of simultaneous access to both ports is described below:

Read-Read Collisions

If both ports read simultaneously from the same memory cell, the read operation is successful.

Write-Write Collisions

If both ports try to write simultaneously to the same memory cell, both outputs are marked as invalid (nan).

Write-Read Collisions

This collision occurs when one port writes and the other reads from the same memory cell. While the memory contents are not corrupted, the validity of the output data on the read port depends on the Write Mode of the write port.

- If the write port is in **Read before write** mode, the other port can reliably read the old memory contents.

- If the write port is in **Read after write** or **No read on write**, data on the output of the read port is invalid (nan).

You can set the Write Mode of each port using the Advanced tab of the block parameters dialog box.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

- **Basic tab:** Parameters specific to the Basic tab are as follows.
  - **Depth:** Specifies the number of words in the memory for Port A, which must be a positive integer. The Port B depth is inferred from the form factor specified by the input data widths.
  - **Initial value vector:** For distributed memory or block RAM, specifies the initial memory contents. The size and precision of the elements of the initial value vector are based on the data format specified for Port A. When the vector is longer than the RAM, the vector's trailing elements are discarded. When the RAM is longer than the vector, the RAM's trailing words are set to zero. The initial value vector is saturated and rounded according to the precision specified on the data port A of RAM.

    Note: UltraRAM memory is initialized to all 0's during power up or device reset. If implemented in UltraRAM, the Single Port RAM block cannot be initialized to user defined values.
  - **Memory Type:** Option to select whether the dual port RAM will be implemented in **Distributed memory**, **Block RAM**, or **UltraRAM**. The distributed dual port RAM is always set to use port A in Read Before Write mode and port B in read-only mode.

    Depending on your selection for **Memory Type**, the dual-port RAM will be inferred or implemented in this way when the design is compiled:
    - If the block will be implemented in **Distributed memory**, the Distributed Memory Generator v8.0 LogiCORE IP will be inferred or implemented when the design is compiled. This LogiCORE IP is described in the *Distributed Memory Generator LogiCORE IP Product Guide* (PG063).
    - If the block will be implemented in block RAM or UltraRAM, the XPM_MEMORY_TDPRAM (True Dual Port RAM) macro will be inferred or implemented when the design is compiled. For information on the XPM_MEMORY_TDPRAM Xilinx Parameterized Macro (XPM), refer to *UltraScale Architecture Libraries Guide* (UG974).
  - **Initial value for port A output Register:** Specifies the initial value for port A output register. The initial value is saturated and rounded according to the precision specified on the data port A of RAM.

    - **Initial value for port B output register:** Specifies the initial value for port B output register. The initial value is saturated and rounded according to the precision specified on the data port B of RAM.
    - **Provide synchronous reset port for port A output register:** When selected, allows access to the reset port available on the port A output register of the block RAM or UltraRAM. The reset port is available only when the latency of the Block RAM or UltraRAM is greater than or equal to 1.
    - **Provide synchronous reset port for port B output register:** When selected, allows access to the reset port available on the port B output register of the Block RAM or UltraRAM. The reset port is available only when the latency of the Block RAM or UltraRAM is greater than or equal to 1.
    - **Provide enable port for port A:** When selected, allows access to the enable port for port A. The enable port is available only when the latency of the block is greater than or equal to 1.
    - **Provide enable port for port B:** When selected, allows access to the enable port for port B. The enable port is available only when the latency of the block is greater than or equal to 1.
- **Advanced tab:** Parameters specific to the Advanced tab are as follows.
    - **Write Modes:**
        - **Port A or Port B:** When the Dual Port RAM block is implemented in block RAM, specifies memory behavior for port A or port B when WE is asserted. Supported modes are: **Read after write**, **Read before write**, and **No read On write**. **Read after write** indicates the output value reflects the state of the memory after the write operation. **Read before write** indicates the output value reflects the state of the memory before the write operation. **No read on write** indicates that the output value remains unchanged irrespective of change of address or state of the memory. There are device specific restrictions on the applicability of these modes. Also refer to the Write Mode topic above for more information.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

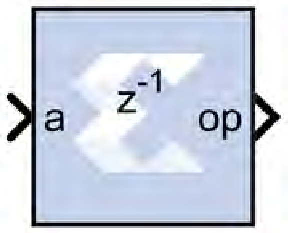
LogiCORE and XPM Documentation

LogiCORE IP Distributed Memory Generator v8.0 (Distributed Memory)

UltraScale Architecture Libraries Guide - XPM_MEMORY_TDPRAM Macro (UltraRAM)

Exponential

This Xilinx Exponential block preforms the exponential operation on the input. Currently, only the floating-point data type is supported.



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:** Parameters specific to the Basic tab are as follows.
  - **AXI Interface:**
    - **Flow Control:**
      - **Blocking:** Selects “Blocking” mode. In this mode, the lack of data on one input channel does block the execution of an operation if data is received on another input channel.
      - **NonBlocking:** Selects “Non-Blocking” mode. In this mode, the lack of data on one input channel does not block the execution of an operation if data is received on another input channel.
    - **Optimize Goal:** When NonBlocking mode is selected, the following optimization options are activated.
      - **Resources:** Block is configured for minimum resources.
      - **Performance:** Block is configured for maximum performance.
  - **Block Memory Usage:**
    - **BMG Usage:**
      - **No Usage:** Do not use Block Memory.
      - **Full Usage:** Make full use of Block Memory.
  - **Latency Specification:**
    - **Latency:** This defines the number of sample periods by which the block's output is delayed.
- **Optional Ports tab:** Parameters specific to the Optional Ports tab are as follows.
  - **Input Channel Ports:**
    - **Has TLAST:** Adds a tlast port to the input channel.

- **Has TUSER:** Adds a tuser port to the input channel.
- **Control Options:**
    - **Provide enable port:** Add an enable port to the block interface.
    - **Has Result TREADY:** Add a TREADY port to the result channel.
- **Exception Signals:**
    - **UNDERFLOW:** Add an output port that serves as an underflow flag.
    - **OVERFLOW:** Add an output port that serves as an overflow flag.

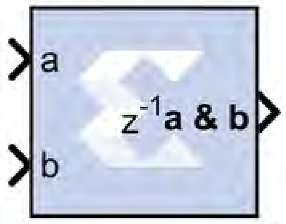
Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

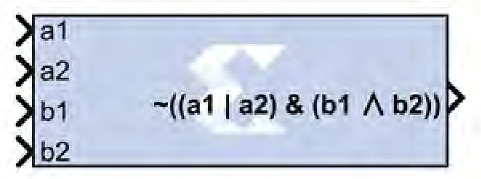
LogiCORE IP Floating-Point Operator v7.1

Expression

The Xilinx Expression block performs a bitwise logical expression.



The expression is specified with operators described in the table below. The number of input ports is inferred from the expression. The input port labels are identified from the expression, and the block is subsequently labeled accordingly. For example, the expression: `~((a1 | a2) & (b1 ^ b2))` results in the following block with 4 input ports labeled `'a1'`, `'a2'`, `'b1'`, and `'b2'`.



The expression is parsed and an equivalent statement is written in VHDL (or Verilog). Shown below, in decreasing order of precedence, are the operators that can be used in the Expression block.

Operator	Symbol
Precedence	()
NOT	~
AND	&
OR	|
XOR	^

Block Parameters

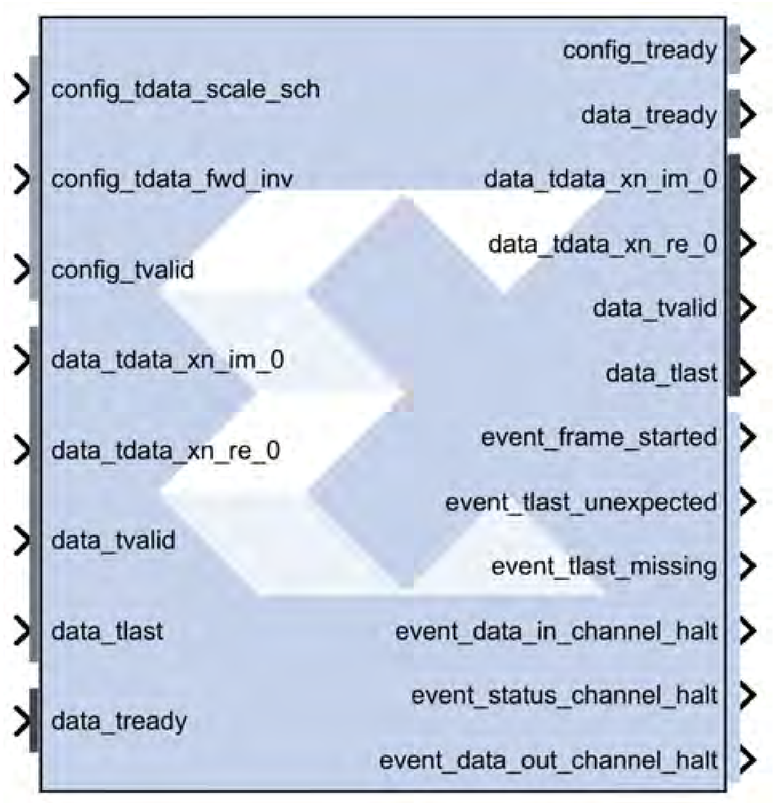
The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:** Parameters specific to the Basic tab are as follows.
  - **Expression:** Bitwise logical expression.
  - **Align Binary Point:** Specifies that the block must align binary points automatically. If not selected, all inputs must have the same binary point position.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Fast Fourier Transform 9.1

The Xilinx Fast Fourier Transform block implements the Cooley-Tukey FFT algorithm, a computationally efficient method for calculating the Discrete Fourier Transform (DFT). In addition, the block provides an AXI4-Stream-compliant interface.



The FFT computes an N-point forward DFT or inverse DFT (IDFT) where, N = 2^m, m = 3 - 16. For fixed-point inputs, the input data is a vector of N complex values represented as dual b_x-bit two's complement numbers, that is, b^x bits for each of the real and imaginary components of the data sample, where b_x is in the range 8 to 34 bit, inclusive. Similarly, the phase factors b_w can be 8 to 34 bits wide.

For single-precision floating-point inputs, the input data is a vector of N complex values represented as dual 32-bit floating-point numbers with the phase factors represented as 24- or 25-bit fixed-point numbers.

Theory of Operation

The FFT is a computationally efficient algorithm for computing a Discrete Fourier Transform (DFT) of sample sizes that are a positive integer power of 2. The DFT of a sequence is defined as:

$$X(k)=\sum_{k=0}^{N-1} x(n)e^{-jnk2\pi/N} \quad k=0,\ldots,N-1$$

where *N* is the transform length and *j* is the square root of -1. The inverse DFT (IDFT) is:

$$x(n) = \frac{1}{N}\sum_{k=0}^{N\ 1} X(k)e^{jnk2\pi/N} \quad n=0,\ldots,N-1$$

AXI Ports that are Unique to this Block

This Sysgen Generator block exposes the AXI CONFIG channel as a group of separate ports based on sub-field names. The sub-field ports are described as follows:

Configuration Channel Input Signals:

config_tdata_scale_sch	A sub-field port that represents the **Scaling Schedule** field in the Configuration Channel vector. Refer to the document LogiCORE IP Fast Fourier Transform v9.1 for an explanation of the bits in this field.
config_tdata_fwd_inv	A sub-field port that represents the **Forward Inverse** field in the Configuration Channel vector. Refer to the document LogiCORE IP Fast Fourier Transform v9.1 for an explanation of the bits in this field.
config_tdata_nfft	A sub-field port that represents the **Transform Size (NFFT)** field in the Configuration Channel vector. Refer to the document LogiCORE IP Fast Fourier Transform v9.1 for an explanation of the bits in this field.
config_tdata_cp_len	A sub-field port that represents the **Cyclic Prefix Length (CP_LEN)** field in the Configuration Channel vector. Refer to the document LogiCORE IP Fast Fourier Transform v9.1 for an explanation of the bits in this field.

This HDL block exposes the AXI DATA channel as separate ports based on the real and imaginary sub-field names. The sub-field ports are described as follows:

DATA Channel Input Signals:

data_tdata_xn_im	Represents the imaginary component of the Data Channel. The signal driving xn_im can be a signed data type of width S with binary point at S-1, where S is a value between 8 and 34, inclusive. eg: Fix_8_7, Fix_34_33. Both xn_re and xn_im signals must have the same data type. Refer to the document LogiCORE IP Fast Fourier Transform v9.1 for an explanation of the bits in this field.
data_tdata_xn_re	Represents the real component of the Data Channel. The signal driving xn_re can be a signed data type of width S with binary point at S-1, where S is a value between 8 and 34, inclusive. eg: Fix_8_7, Fix_34_33. Both xn_re and xn_im signals must have the same data type. Refer to the document LogiCORE IP Fast Fourier Transform v9.1 for an explanation of the bits in this field.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:** Parameters specific to the Basic tab are as follows.

- **Transform Length:**
    - **Transform_length:** One of N = $2^{(3..16)}$ = 8 - 65536.
- **Architecture Configuration:**
    - **Target Clock Frequency(MHz):** Enter the target clock frequency.
    - **Target Data Throughput(MSPS):** Enter the target throughput.
    - **Architecture Choice:** Choose one of the following.
        - **automatically_select**
        - **pipelined_streaming_io**
        - **radix_4_burst_io**
        - **radix_2_burst_io**
        - **radix_2_lite_burst_io**
- **Transform Length Options:**
    - **Run Time Configurable Transform Length:** The transform length can be set through the nfft port if this option is selected. Valid settings and the corresponding transform sizes are provided in the section titled Transform Size in the associated document LogiCORE IP Fast Fourier Transform v9.1 for an explanation of the bits in this field.

- **Advanced tab:** Parameters specific to the Advanced tab are as follows.
    - **Precision Options:**
        - **Phase Factor Width:** Choose a value between 8 and 34, inclusive to be used as bit widths for phase factors.
    - **Scaling Options:** Select between **Unscaled**, **Scaled**, and **Block Floating Point** output data types.
        - **Rounding Modes:**
            - **Truncation:** To be applied at the output of each rank.
        - **Convergent Rounding:** To be applied at the output of each rank.
    - **Control Signals:**
        - **ACLKEN:** Enables the clock enable (aclken) pin on the core. All registers in the core are enabled by this control signal.
        - **ARESETn:** Active-low synchronous clear input that always takes priority over ACLKEN. A minimum ARESETn active pulse of two cycles is required, since the signal is internally registered for performance. A pulse of one cycle resets the core, but the response to the pulse is not in the cycle immediately following.

- **Output Ordering:**
    - **Cyclic Prefix Insertion:**

      Cyclic prefix insertion takes a section of the output of the FFT and prefixes it to the beginning of the transform. The resultant output data consists of the cyclic prefix (a copy of the end of the output data) followed by the complete output data, all in natural order. Cyclic prefix insertion is only available when output ordering is Natural Order.

      When cyclic prefix insertion is used, the length of the cyclic prefix can be set frame-by-frame without interrupting frame processing. The cyclic prefix length can be any number of samples from zero to one less than the point size. The cyclic prefix length is set by the CP_LEN field in the Configuration channel. For example, when N = 1024, the cyclic prefix length can be from 0 to 1023 samples, and a CP_LEN value of 0010010110 produces a cyclic prefix consisting of the last 150 samples of the output data.

    - **Output ordering:** Choose between **Bit/Digit Reversed Order** or **Natural Order** output.
- **Throttle Schemes:** Select the tradeoff between performance and data timing requirements.
    - **Real Time:** This mode typically gives a smaller and faster design, but has strict constraints on when data must be provided and consumed.
    - **Non Real Time:** This mode has no such constraints, but the design might be larger and slower.
- **Optional Output Fields:**
    - **XK_INDEX:** The XK_INDEX field (if present in the Data Output channel) gives the sample number of the XK_RE/XK_IM data being presented at the same time. In the case of natural order outputs, XK_INDEX increments from 0 to (point size) -1. When bit reversed outputs are used, XK_INDEX covers the same range of numbers, but in a bit (or digit) reversed manner.
    - **OVFLO:**

      The Overflow (OVFLO) field in the Data Output and Status channels is only available when the Scaled arithmetic is used. OVFLO is driven High during unloading if any point in the data frame overflowed.

      For a multichannel core, there is a separate OVFLO field for each channel. When an overflow occurs in the core, the data is wrapped rather than saturated, resulting in the transformed data becoming unusable for most applications

- **Block Icon Display:**
    - **Display shortened port names:** On by default. When unchecked, **data_tvalid**, for example, becomes **m_axis_data_tvalid**.

- **Implementation tab:** Parameters specific to the Implementation tab are as follows.

- **Memory Options:**
  - **Data:** Option to choose between **Block RAM** and **Distributed RAM**. This option is available only for sample points 8 through 1024. This option is not available for Pipelined Streaming I/O implementation.
  - **Phase Factors:** Choose between **Block RAM** and **Distributed RAM**. This option is available only for sample points 8 till 1024. This option is not available for Pipelined Streaming I/O implementation.
  - **Number Of Stages Using Block RAM:** Store data and phase factor in **Block RAM** and partially in **Distributed RAM**. This option is available only for the Pipelined Streaming I/O implementation.
  - **Reorder Buffer:** Choose between **Block RAM** and **Distributed RAM** up to 1024 points transform size.
  - **Hybrid Memories:** Click check box to **Optimize Block RAM Count Using Hybrid Memories**.
- **Optimize Options:**
  - **Complex Multipliers:** Choose one of the following.
    - Use CLB logic
    - Use 3-multiplier structure (resource optimization)
    - Use 4-multiplier structure (performance optimization)
  - **Butterfly Arithmetic:** Choose one of the following:
    - Use CLB logic
    - Use XTremeDSP Slices

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Block Timing

To better understand the FFT blocks control behavior and timing, please consult the core data sheet.

LogiCORE Documentation

LogiCORE IP Fast Fourier Transform v9.1

LogiCORE IP Floating-Point Operator v7.1

FDATool

The Xilinx FDATool block provides an interface to the FDATool software available as part of the MATLAB® Signal Processing Toolbox.

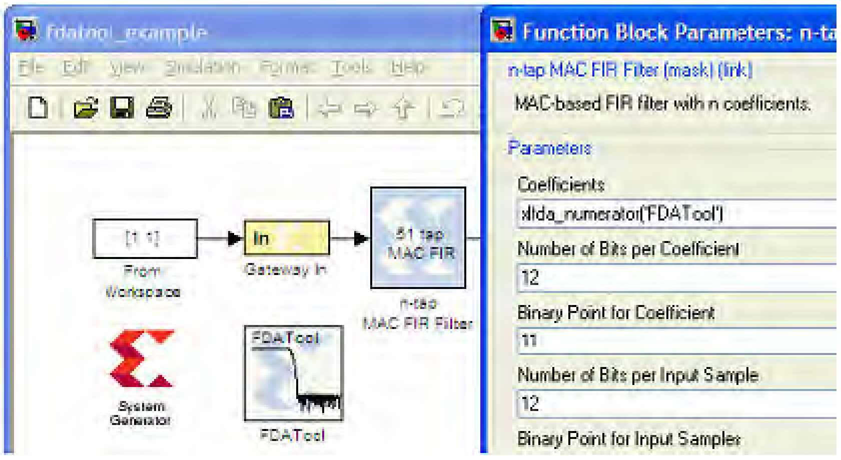
The block does not function properly and should not be used if the Signal Processing Toolbox is not installed. This block provides a means of defining an FDATool object and storing it as part of a Model Composer model. FDATool provides a powerful means for defining digital filters with a graphical user interface.

Example of Use

Copy an FDATool block into a Subsystem where you would like to define a filter. Double-clicking the block icon opens up an FDATool session and graphical user interface. The filter is stored in an data structure internal to the FDATool interface block, and the coefficients can be extracted using MATLAB® helper functions provided as part of Model Composer. The function call `xlfda_numerator('FDATool')` returns the numerator of the transfer function (e.g., the impulse response of a finite impulse response filter) of the FDATool block named `'FDATool'`. Similarly, the helper function `xlfda_denominator('FDATool')` retrieves the denominator for a non-FIR filter.

A typical use of the FDATool block is as a companion to an FIR filter block, where the Coefficients field of the filter block is set to `xlfda_numerator('FDATool')`. An example is shown in the following diagram:

Figure 330: **FDATool Example**



n-tap MAC FIR Filter (mask) (link)

MAC-based FIR filter with n coefficients.

Parameters

Coefficients: xlfda_numerator('FDATool')

Number of Bits per Coefficient: 12

Binary Point for Coefficient: 11

Number of Bits per Input Sample: 12

Binary Point for Input Samples

Note that `xlfda_numerator()` can equally well be used to initialize a memory block or a `'coefficient'` variable for a masked Subsystem containing an FIR filter.

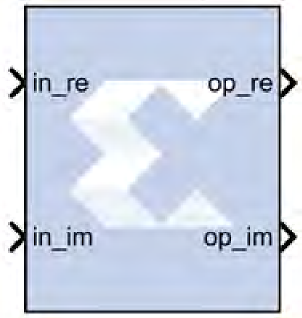
This block does not use any hardware resources.

FDA Tool Interface

Double-clicking the icon in your Simulink model opens up an FDATool session and its graphical user interface. Upon closing the FDATool session, the underlying FDATool object is stored in the UserData parameter of the Xilinx FDATool block. Use the `xlfda_numerator()` helper function and `get_param()` to extract information from the object as desired.

FFT

The Xilinx FFT (Fast Fourier Transform) block takes a block of time domain waveform data and computes the frequency of the sinusoid signals that make up the waveform.



FFT is a fast implementation of the discrete Fourier transform. The data of the time domain signal is sampled at discrete intervals. The sampling frequency is twice the maximum frequency that can be resolved by the FFT, based on the Nyquist theorem. If a signal is sampled at 1 kHz, the highest frequency that can be resolved by the FFT is 500 Hz.

$f_s = f_{max}/2$

where f_{max} = maximum resolvable frequency and f_s = sampling frequency.

The duration of the data sample is inversely proportional to the frequency resolution of the FFT. The longer the sample duration, the higher the number of data points, and the finer the frequency resolution. If a signal sampled at f_s for twice the duration, the difference between successive frequency d_f is halved, resulting in an FFT with finer frequency resolution.

$d_f = 1/T$

where d_f = frequency resolution of the FFT, and T= total sampling time.

The number of samples taken over time T is N, so sampling frequency is N/T samples/sec.

Description

FFT is a computationally efficient implementation of the Discrete Fourier Transform (DFT). A DFT is a collection of data points detailing the correlation between the time domain signal and sinusoids at discrete frequencies.

The DFT is defined by the following equation:

$$X(k)=\sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}nk} \quad for \quad k=0,1,2,\ldots,N-1$$

where *N* is the transform length, *k* is used to denote the frequency domain ordinal, and *n* is used to represent the time-domain ordinal.

The FFT block is ideal for implementing simple Fourier transforms. If your FFT implementation will use more complicated transform features such as an AXI4-Stream-compliant interface, a real time throttle scheme, Radix-4 Burst I/O, or Radix-2 Lite Burst I/O, use the Xilinx Fast Fourier Transform 9.1 block in your design instead of the FFT block.

In the Vivado® design flow, the FFT block is inferred as "LogiCORE IP Fast Fourier Transform v9.1" for code generation. Refer to the document LogiCORE IP Fast Fourier Transform v9.1 for details on this LogicCore IP.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

Parameters specific to the Xilinx FFT block are as follows.

- **Transform Length:** Select the desired point size ranging from 8 to 65536.
- **Scale Result by FFT length:** If selected, data is scaled between FFT stages using a scaling schedule determined by the **Transform Length** setting. If not selected, data is unscaled, and all integer bit growth is carried to the output.
- **Natural Order:** If selected, the output of the FFT block will be ordered in natural order. If not selected, the output of the FFT block will be ordered in bit/digit reversed order.
- **Optimize for:** Directs the block to be optimized for either speed (**Performance**) or area (**Resources**) in the generated hardware.

  Note: If **Resources** is selected and the input sample period is 8 times slower than the system sample period, the block implements Radix-2 Burst I/O architecture. Otherwise, Pipeline Streaming I/O architecture will be used.
- **Optional Port:**
  - **Provide start frame port:** Adds `start_frame_in` and `start_frame_out` ports to the block. The signals on these ports can be used to synchronize frames at the input and output of the FFT block. See Adding Start Frame Ports to Synchronize Frames for a description of the operation of these two ports.

Context Based Pipeline vs. Radix Implementation

Pipelined Streaming I/O and Radix-2 Burst I/O architectures are supported by the FFT block. Radix-4 Burst I/O architecture is implemented when you select **Optimize for: Resources** block parameter and the sample rate of the inputs is 8 times slower than the system rate. In all other configurations Pipelined Streaming I/O architecture is implemented by default.

Input Data Type Support

The FFT block accepts inputs of varying bit widths with changeable binary point location, such as Fix_16_0 or Fix_30_10, etc. in unscaled block configuration. For the scaled configuration, the input is supported in the same format as the Fast Fourier Transform 9.1 block. The Fast Fourier Transform 9.1 block accepts input values only in the normalized form in the format of Fix_*x*_[*x*-1] (for example, Fix_16_15), so the inputs are 2's complement with a single sign/integer bit.

Latency Value Displayed on the Block

The latency value depends on parameters selected by the user, and the corresponding latency value is displayed on the FFT block icon in the Simulink model.

Automatic Fixed Point and Floating Point Support

Signed fixed point and floating point data types are supported.

For floating point input, either scaled or unscaled data can be selected in the FFT block parameters. In the Fast Fourier Transform 9.1 block, the floating point data type is accepted only when the scaled configuration is selected by the user.

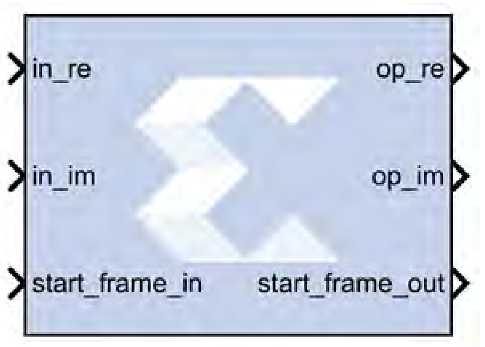
Handling Overflow for Scaled Configuration

The FFT block uses a conservative schedule to avoid overflow scenarios. This schedule sets the scaling value for the corresponding FFT stages in a way that makes sure no overflow occurs.

Adding Start Frame Ports to Synchronize Frames

Selecting **Provide start frame port** in the FFT block properties dialog box adds `start_frame_in` and `start_frame_out` ports at the input and output of the FFT block. These ports are used to synchronize frames at the input and output of the FFT block.

Figure 331: **Adding Start Frame Ports**



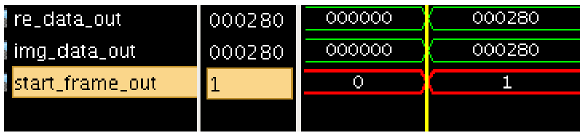
You must provide a valid input at the `start_frame_in` port. When the `start_frame_in` signal is asserted, an impulse is generated at the start of every frame to signal the FFT block to start processing the frame. The frame size is the **Transform Length** entered in the block parameters dialog box.

The `start_frame_out` port provides the information as to when the output frames start. An impulse at the start of every frame on the output side helps in tracking the block behavior.

The FFT block has a frame alignment requirement and these ports help the block operate in accordance with this requirement.

The figure below shows that as soon as the output is processed by the FFT block the `start_frame_out` signal becomes High (1).

Figure 332: **Output**



The following apply to the **Provide start frame port** option and the start frame ports added to the FFT block when the option is enabled:

- The **Provide start frame port** option selection is valid only for Pipelined Streaming I/O architecture. See Context Based Pipeline vs. Radix Implementation for a description of the conditions under which Pipelined Streaming I/O architecture is implemented.
- The option is valid only for input of type fixed point.
- Verilog is supported for netlist generation currently, when the **Provide start frame port** option is selected.

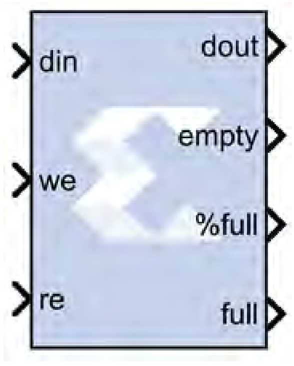
***Note*:** The first sample input to the FFT block may be ignored and users are advised to drive the input data accordingly.

LogiCORE Documentation

LogiCORE IP Fast Fourier Transform v9.1

FIFO

The Xilinx FIFO block implements an FIFO memory queue.



Values presented at the module's data-input port are written to the next available empty memory location when the write-enable input is one. By asserting the read-enable input port, data can be read out of the FIFO using the data output port (`dout`) in the order in which they were written. The FIFO can be implemented using block RAM, distributed RAM, SRL, or built-in FIFO.

The `full` output port is asserted to one when no unused locations remain in the module's internal memory. The `percent_full` output port indicates the percentage of the FIFO that is full, represented with user-specified precision. When the `empty` output port is asserted the FIFO is empty.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows.

  - **FIFO Implementation:**
    - **Memory Type:**

      This block implements FIFOs built from block RAM, distributed RAM, shift registers, or the 7 series built-in FIFOs. Memory primitives are arranged in an optimal configuration based on the selected width and depth of the FIFO. The following table provides best-use recommendations for specific design requirements.

Table 54: **Memory Type**

	Independent Clocks	Common Clock	Small Buffering	Medium-Large Buffering	High Performance	Minimal Resources
7 Series, with Built-In FIFO	X	X		X	X	X
Block RAM	X	X		X	X	X
Shift Register		X	X		X	
Distributed RAM	X	X	X		X	

    - **Performance Options:**
      - **Standard FIFO:** FIFO will operate in Standard Mode.
      - **First Word Fall Through:** FIFO will operate in First-Word Fall-Through (FWFT) mode. The First-Word Fall-Through feature provides the ability to look-ahead to the next word available from the FIFO without issuing a read operation. When data is available in the FIFO, the first word falls through the FIFO and appears automatically on the output. FWFT is useful in applications that require low-latency access to data and to applications that require throttling based on the contents of the data that are read. FWFT support is included in FIFOs created with block RAM, distributed RAM, or built-in FIFOs in 7 series devices.

        - **Implementation Options:**
            - **Use Embedded Registers (when possible):** In 7 series FPGA block RAM and FIFO macros, embedded output registers are available to increase performance and add a pipeline register to the macros. This feature can be leveraged to add one additional cycle of latency to the FIFO core (DOUT bus and VALID outputs) or implement the output registers for FWFT FIFOs. The embedded registers available in 7 series FPGAs can be reset (DOUT) to a default or user programmed value for common clock built-in FIFOs. See the topic **Embedded Registers in block RAM and FIFO Macros** in the LogiCORE IP FIFO Generator 12.0.

- **Depth:** Specifies the number of words that can be stored. Range 16-4M.
- **Specify custom dout width:** Allows specification of asymmetric data widths for input and output. When this option is enabled, the input `din` must be of type Unsigned int.
    - **Dout Width:** Specifies the custom width of `dout` . The default value is `32`. The ratio between width of `din` and `dout` should be one of 1:1, 1:2, 1: 4, 1:8, 2:1, 4:1, or 8:1.

      For example, if the width of the `din` is `32`. Then the `dout` can be 4, 8, 16, 32, 64, 128, or 256.

  IMPORTANT! *Asymmetric data-widths are supported only for Versal devices with Block RAM configuration.*

  IMPORTANT! *The total size of the memory (depth*width) should not exceed 150M.*

- **Bits of precision to use for %full signal:** Specifies the bit width of the %full port. The binary point for this unsigned output is always at the top of the word. Thus, if for example precision is set to one, the output can take two values: 0.0 and 0.5, the latter indicating the FIFO is at least 50% full.
- **Optional Ports:**
    - **Provide reset port:** Add a reset port to the block.
        - **Reset Latency:** Creates a latency on the reset by adding registers. The default is 1.

          Note: For UltraScale™ devices, after the reset gets asserted, the FIFO will remain disable for the next 20 cycles. During this 20 cycle period, all read and write operations are ignored.
        - **Provide enable port:** Add enable port to the block.
        - **Provide data count port:** Add data count port to the block. Provides the number of words in the FIFO.
        - **Provide percent full port:** Add a percent full output port to the block. Indicates the percentage of the FIFO that is full using the user-specified precision. This optional port is turned on by default for backward compatibility reasons.

- **Provide almost empty port:** Add almost empty (ae) port to the block.
- **Provide almost full port:** Add almost efull (af) port to the block.

Following are some general guidelines to use Reset, Write enable, Read enable for the 'built-in FIFO' Memory type:

- **7 series devices:** Without Reset port, it is required to run at least 8 clock cycles latency before asserting WE/RE signals.With Reset port, it is required to run Reset signal ON for at least three clock cycles. During this time no WE or RE signals should be asserted. To be consistent across all built-in FIFO configurations, it is recommended to give reset pulse of at least five clock cycles.

  After Reset de-assertion, run at least 30 clock cycles (reset duration +30 clock cycles duration is defined as a no access zone) before asserting WE/RE signals.
- **UltraScale devices:** The built-in FIFO requires a reset pulse of at least one clock cycle.
- **Versal Devices:** Read enable (`rd_en`) and Write enable (`wr_en`) signals can be made high only when `rd_rst_busy` and `wr_rst_busy` signals are low.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

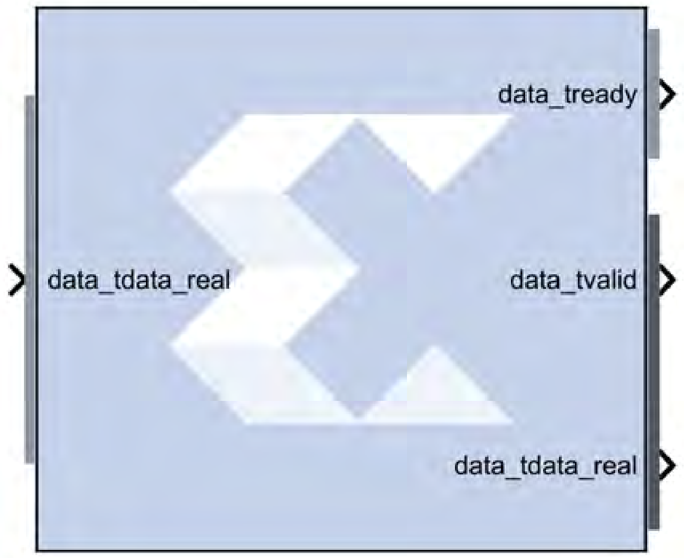
LogiCORE Documentation

LogiCORE IP FIFO Generator 12.0

LogiCORE IP Floating-Point Operator v7.1

FIR Compiler 7.2

This Xilinx FIR Compiler block provides users with a way to generate highly parameterizable, area-efficient, high-performance FIR filters with an AXI4-Stream-compliant interface.



AXI Ports that are Unique to this Block

This block exposes the AXI CONFIG channel as a group of separate ports based on sub-field names. The sub-field ports are described as follows:

Configuration Channel Input Signals:

config_tdata_fsel	A sub-field port that represents the **fsel** field in the Configuration Channel vector. **fsel** is used to select the active filter set. This port is exposed when the number of coefficient sets is greater than one. Refer to the FIR Compiler V7.2 Product Guide for an explanation of the bits in this field.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Filter Specification tab:**

  Parameters specific to the Filter Specification tab are as follows.

  - **Filter Coefficients:**

    - **Coefficient Vector:** Specifies the coefficient vector as a single MATLAB® row vector. The number of taps is inferred from the length of the MATLAB® row vector. If multiple coefficient sets are specified, then each set is appended to the previous set in the vector. It is possible to enter these coefficients using the FDATool block as well.

    - **Number of Coefficients Sets:** The number of sets of filter coefficients to be implemented. The value specified must divide without remainder into the number of coefficients.

- **Use Reloadable Coefficients:** Check to add the coefficient reload ports to the block. The set of data loaded into the reload channel will not take action until triggered by a re-configuration synchronization event. Refer to the FIR Compiler V7.2 Product Guide for a more detailed explanation of the RELOAD Channel interface timing. This block supports the xlGetReloadOrder function. See the Model Composer Utility function xlGetReloadOrder for details.

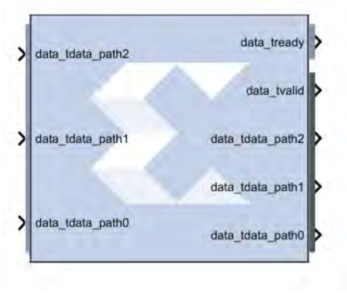
- **Filter Specification:**
  - **Filter Type:**
    - **Single_Rate:** The data rate of the input and the output are the same.
    - **Interpolation:** The data rate of the output is faster than the input by a factor specified by the Interpolation Rate value.
    - **Decimation:** The data rate of the output is slower than the input by a factor specified in the Decimation Rate Value.
    - **Hilbert:** Filter uses the Hilbert Transform.
    - **Interpolated:** An interpolated FIR filter has a similar architecture to a conventional FIR filter, but with the unit delay operator replaced by k-1 units of delay. k is referred to as the zero-packing factor. The interpolated FIR should not be confused with an interpolation filter. Interpolated filters are single-rate systems employed to produce efficient realizations of narrow-band filters and, with some minor enhancements, wide-band filters can be accommodated. The data rate of the input and the output are the same.
  - **Rate Change Type:** This field is applicable to Interpolation and Decimation filter types. Used to specify an **Integer** or **Fixed_Fractional** rate change.
  - **Interpolation Rate Value:** This field is applicable to all Interpolation filter types and Decimation filter types for Fractional Rate Change implementations. The value provided in this field defines the up-sampling factor, or P for Fixed Fractional Rate (P/Q) resampling filter implementations.
  - **Decimation Rate Value:** This field is applicable to the all Decimation and Interpolation filter types for Fractional Rate Change implementations. The value provided in this field defines the down-sampling factor, or Q for Fixed Fractional Rate (P/Q) resampling filter implementations.
  - **Zero pack factor:** Allows you to specify the number of 0's inserted between the coefficient specified by the coefficient vector. A zero packing factor of k inserts k-1 0s between the supplied coefficient values. This parameter is only active when the Filter type is set to Interpolated.

- **Channel Specification tab:** Parameters specific to the Channel Specification tab are as follows.
  - **Interleaved Channel Specification:**

- **Channel Sequence:** Select Basic or Advanced. See the LogiCORE IP FIR Compiler v7.2 Product Guide for an explanation of the advanced channel specification feature.
- **Number of Channels:** The number of data channels to be processed by the FIR Compiler block. The multiple channel data is passed to the core in a time-multiplexed manner. A maximum of 64 channels is supported.
- **Sequence ID List:** A comma delimited list that specifies which channel sequences are implemented.

- **Parallel Channel Specification:**
  - **Number of Paths:** Specifies the number of parallel data paths the filter is to process. As shown below, when more than one path is specified, the data_tdata input port is divided into sub-ports that represent each parallel path.

Figure 333: **Number of Paths**



- **Hardware Oversampling Specification:**
  - **Select format:**
    - **Maximum_Possible:** Specifies that oversampling be automatically determined based on the din sample rate.
    - **Input_Sample_Period/Output_Sample_Period:** Activates the Sample period dialog box below. Enter the Sample Period specification. Selecting this option exposes the s_axis_data_tvalid port (called ND port on earlier versions of the core). With this port exposed, no input handshake abstraction and no rate-propagation takes place.
    - **Hardware Oversampling Rate:** Activates the Hardware Oversampling Rate dialog box. Enter the Hardware Oversampling Rate specification below.
- **Hardware Oversampling Rate:** The hardware oversampling rate determines the degree of parallelism. A rate of one produces a fully parallel filter. A rate of n (resp., n+1) for an n-bit input signal produces a fully serial implementation for a non-symmetric (resp., symmetric) impulse response. Intermediate values produce implementations with intermediate levels of parallelism.

- **Implementation tab:** Parameters specific to the Implementation tab are as follows.

- **Coefficient Options:**
  - **Coefficient Type:** Specify Signed or Unsigned.
  - **Quantization:** Specifies the quantization method to be used for quantizing the coefficients. This can be set to one of the following:
    - Integer_Coefficients
    - Quantize_Only
    - Maximize_Dynamic_Range
    - Normalize_to_Centre_Coefficient
  - **Coefficient Width:** Specifies the number of bits used to represent the coefficients.
  - **Best Precision Fractional Bits:** When selected, the coefficient fractional width is automatically set to maximize the precision of the specified filter coefficients.
  - **Coefficient Fractional Bits:** Specifies the binary point location in the coefficients datapath options.
  - **Coefficients Structure:** Specifies the coefficient structure. Depending on the coefficient structure, optimizations are made in the core to reduce the amount of hardware required to implement a particular filter configuration. The selected structure can be any of the following.
    - Inferred
    - Non-Symmetric
    - Symmetric
    - Negative_Symmetric
    - Half_Band
    - Hilbert

    The vector of coefficients specified must match the structure specified unless Inferred from coefficients is selected in which case the structure is determined automatically from these coefficients.
- **Datapath Options:**
  - **Output Rounding Mode:** Choose one of the following.
    - Full_Precision
    - Truncate_LSBs
    - Non_Symmetric_Rounding_Down
    - Non_Symmetric_Rounding_Up

        - Symmetric_Rounding_to_Zero
        - Symmetric_Rounding_to_Infinity
        - Convergent_Rounding_to_Even
        - Convergent_Rounding_to_Odd
    - **Output Width:** Specify the output width. Edit box activated only if the Rounding mode is set to a value other than Full_Precision.
- **Detailed Implementation tab:** Parameters specific to the Detailed Implementation tab are as follows.
    - **Filter Architecture:** The following two filter architectures are supported.
        - Systolic_Multiply_Accumulate
        - Transpose_Multiply_Accumulate

        ***Note*:** When selecting the Transpose Multiply-Accumulate architecture, these limitations apply:

            - Symmetry is not exploited. If the **Coefficient Vector** specified on the Filter Specification tab is detected as symmetric, the FIR Compiler 7.2 block parameters dialog box will not allow you to select Transpose Multiply Accumulate.
            - Multiple interleaved channels are not supported.
    - **Optimization Options:**

        Specifies if the core is required to operate at maximum possible speed ("Speed" option) or minimum area ("Area" option). The "Area" option is the recommended default and will normally achieve the best speed and area for the design, however in certain configurations, the "Speed" setting might be required to improve performance at the expense of overall resource usage (this setting normally adds pipeline registers in critical paths).

        - **Goal:**
            - Area
            - Speed
            - Custom
        - **List:** A comma delimited list that specifies which optimizations are implemented by the block. The optimizations are as follows.
            - **Data_Path_Fanout:** Adds additional pipeline registers on the data memory outputs to minimize fan-out. Useful when implementing large data width filters requiring multiple DSP slices per multiply-add unit.
            - **Pre-Adder_Pipeline:** Pipelines the pre-adder when implemented using fabric resources. This may occur when a large coefficient width is specified.

  - **Coefficient_Fanout:** Adds additional pipeline registers on the coefficient memory outputs to minimize fan-out. Useful for Parallel channels or large coefficient width filters requiring multiple DSP slices per multiply-add unit.
  - **Control_Path_Fanout:** Adds additional pipeline registers to control logic when Parallel channels have been specified.
  - **Control_Column_Fanout:** Adds additional pipeline registers to control logic when multiple DSP columns are required to implement the filter.
  - **Control_Broadcast_Fanout:** Adds additional pipeline registers to control logic for fully parallel (one clock cycle per channel per input sample) symmetric filter implementations.
  - **Control_LUT_Pipeline:** Pipelines the Look-up tables required to implement the control logic for Advanced Channel sequences.
  - **No_BRAM_Read_First_Mode:** Specifies that Block RAM READ-FIRST mode should not be used.
  - **Increased speed:** Multiple DSP slice columns are required for non-symmetric filter implementations.
  - **Other:** Miscellaneous optimizations.

  Note: All optimizations maybe specified but are only implemented when relevant to the core configuration.

- **Memory Options:** The memory type for MAC implementations can either be user-selected or chosen automatically to suit the best implementation options. Note that a choice of "Distributed" might result in a shift register implementation where appropriate to the filter structure. Forcing the RAM selection to be either Block or Distributed should be used with caution, as inappropriate use can lead to inefficient resource usage - the default Automatic mode is recommended for most applications.
  - **Data Buffer Type:** Specifies the type of memory used to store data samples.
  - **Coefficient Buffer Type:** Specifies the type of memory used to store the coefficients.
  - **Input Buffer Type:** Specifies the type of memory to be used to implement the data input buffer, where present.
  - **Output Buffer type:** Specifies the type of memory to be used to implement the data output buffer, where present.
  - **Preference for other storage:** Specifies the type of memory to be used to implement general storage in the datapath.
- **DSP Slice Column Options:**

    - **Multi-Column Support:** For device families with DSP slices, implementations of large high speed filters might require chaining of DSP slice elements across multiple columns. Where applicable (the feature is only enabled for multi-column devices), you can select the method of folding the filter structure across the multiple-columns, which can be Automatic (based on the selected device for the project) or Custom (you select the length of the first and subsequent columns).
    - **Column Configuration:** Specifies the individual column lengths in a comma delimited list. (See the data sheet for a more detailed explanation.)
    - **Inter-Column Pipe Length:** Pipeline stages are required to connect between the columns, with the level of pipelining required being depending on the required system clock rate, the chosen device and other system-level parameters. The choice of this parameter is always left for you to specify.

- **Interface tab:**
  - **Data Channel Options:**
    - **TLAST:** TLAST can either be Not_Required, in which case the block will not have the port, or Vector_Framing, where TLAST is expected to denote the last sample of an interleaved cycle of data channels, or Packet_Framing, where the block does not interpret TLAST, but passes the signal to the output DATA channel TLAST with the same latency as the datapath.
    - **Output TREADY:** This field enables the data_tready port. With this port enabled, the block will support back-pressure. Without the port, back-pressure is not supported, but resources are saved and performance is likely to be higher.
    - **Input FIFO:** Selects a FIFO interface for the S_AXIS_DATA channel. When the FIFO has been selected, data can be transferred in a continuous burst up to the size of the FIFO (default 16) or, if greater, the number of interleaved data channels. The FIFO requires additional FPGA logic resources.
    - **TUSER:** Select one of the following options for the Input and the Output.
      - **Not_Required:** Neither of the uses is required; the channel in question will not have a TUSER field.
      - **User_Field:** In this mode, the block ignores the content of the TUSER field, but passes the content untouched from the input channel to the output channels.
      - **Chan_ID_Field:** In this mode, the TUSER field identifies the time-division-multiplexed channel for the transfer.
      - **User and Chan_ID_Field:** In this mode, the TUSER field will have both a user field and a chan_id field, with the chan_id field in the least significant bits. The minimal number of bits required to describe the channel will determine the width of the chan_id field, e.g. 7 channels will require 3 bits.
  - **Configuration Channel Options:**

- **Synchronization Mode:**
  - **On_Vector:** Configuration packets, when available, are consumed and their contents applied when the first sample of an interleaved data channel sequence is processed by the block. When the block is configured to process a single data channel configuration packets are consumed every processing cycle of the block.
  - **On_Packet:** Further qualifies the consumption of configuration packets. Packets will only be consumed once the block has received a transaction on the s_axis_data channel where s_axis_data_tlast has been asserted.
- **Configuration Method:**
  - **Single:** A single coefficient set is used to process all interleaved data channels.
  - **By_Channel:** A unique coefficient set is specified for each interleaved data channel.

- **Reload Channel Options:**
  - **Reload Slots:** Specifies the number of coefficient sets that can be loaded in advance. Reloaded coefficients are only applied to the block once the configuration packet has been consumed. (Range 1 to 256).
- **Control Options:**
  - **ACLKEN:** Active-high clock enable. Available for MAC-based FIR implementations.
  - **ARESETn (active low):** Active-low synchronous clear input that always takes priority over ACLKEN. A minimum ARESETn active pulse of two cycles is required, since the signal is internally registered for performance. A pulse of one cycle resets the control and datapath of the core, but the response to the pulse is not in the cycle immediately following.

- **Advanced tab:**
  - **Block Icon Display:**
    - **Display shortened port names:**

      On by default. When unchecked, data_tvalid, for example, becomes m_axis_data_tvalid.

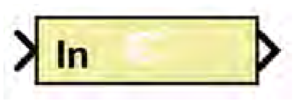
      Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP FIR Compiler v7.2

Gateway In

The Xilinx Gateway In blocks are the inputs into the HDL portion of your Simulink® design. These blocks convert Simulink® integer, double, and fixed-point data types into the Model Composer fixed-point type. Each block defines a top-level input port or interface in the HDL design generated by Model Composer.



Conversion of Simulink Data to Model Composer Data

A number of different Simulink data types are supported on the input of Gateway In. The data types supported include int8, uint8, int16, uint16, in32, uint32, single, double, and Simulink fixed point data type (if Simulink fixed point data type license is available). In all causes the input data is converted to a double internal to gateway and then converted to target data type as specified on the Gateway In block (Fixed Point, Floating Point or Boolean). When converting to Fixed point from the internal double representation, the Quantization and Overflow is further handled as specified in the Block GUI. For overflow, the options are to saturate to the largest positive/ smallest negative value, to wrap (for example, to discard bits to the left of the most significant representable bit), or to flag an overflow as a Simulink error during simulation. For quantization, the options are to round to the nearest representable value (or to the value furthest from zero if there are two equidistant nearest representable values), or to truncate (for example, to discard bits to the right of the least significant representable bit).It is important to realize that conversion, overflow and quantization do not take place in hardware, they take place only in the simulation model of the block.

Gateway Blocks

As listed below, the Xilinx *Gateway In* block is used to provide a number of functions:

- Converting data from Simulink integer, double and fixed-point type to the Model Composer fixed-point type during simulation in Simulink.
- Defining top-level input ports or interface in the HDL design generated by Model Composer.
- Defining test bench stimuli when the **Create Testbench** box is checked in the System Generator token. In this case, during HDL code generation, the inputs to the block that occur during Simulink simulation are logged as a logic vector in a data file. During HDL simulation, an entity that is inserted in the top level test bench checks this vector and the corresponding vectors produced by Gateway Out blocks against expected results.
- Naming the corresponding port in the top level HDL entity.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

- **Basic tab:**

Parameters specific to the Basic tab are as follows.

- **Output Type:**

  Specifies the output data type. Can be **Boolean**, **Fixed-point**, or **Floating-point**.

- **Arithmetic Type:** If the Output Type is specified as Fixed-point, you can select **Signed (2's comp)** or **Unsigned** as the Arithmetic Type.

- **Fixed-point Precision:**

  - **Number of bits:** Specifies the total number of bits, including the binary point bit width.

  - **Binary point:** Specifies the bit location of the binary point, where bit zero is the least significant bit.

- **Floating-point Precision:**

  - **Single:** Specifies single precision (32 bits).

  - **Double:** Specifies double precision (64 bits).

  - **Custom:** Activates the field below so you can specify the Exponent width and the Fraction width.

  - **Exponent width:** Specify the exponent width.

  - **Fraction width:** Specify the fraction width.

- **Quantization:**

  Quantization errors occur when the number of fractional bits is insufficient to represent the fractional portion of a value. The options are to **Truncate** (for example, to discard bits to the right of the least significant representable bit), or to **Round(unbiased: +/- inf)** or **Round (unbiased: even values).**

  - **Round(unbiased: +/- inf):**

    Also known as "Symmetric Round (towards +/- inf)" or "Symmetric Round (away from zero)". This is similar to the MATLAB® `round()` function. This method rounds the value to the nearest desired bit away from zero and when there is a value at the midpoint between two possible rounded values, the one with the larger magnitude is selected. For example, to round 01.0110 to a Fix_4_2, this yields 01.10, since 01.0110 is exactly between 01.01 and 01.10, and the latter is further from zero.

- **Overflow:**

Overflow errors occur when a value lies outside the representable range. For overflow the options are to **Saturate** to the largest positive/smallest negative value, to **Wrap** (for example, to discard bits to the left of the most significant representable bit), or to **Flag as error** (an overflow as a Simulink® error) during simulation. **Flag as error** is a simulation only feature. The hardware generated is the same as when **Wrap** is selected.

- **Implementation tab:**

  Parameters specific to the Implementation tab are as follows.

  - **Interface Options:**
    - **Interface:**
      - **None:** Implies that during HDL Netlist generation, this Gateway In will be translated as an Input Port at the top level.
      - **AXI4-Lite:** Implies that during HDL Netlist generation, an AXI4-Lite interface will be created and this Gateway In will be mapped to one of the registers within the AXI4-Lite interface.
    - **Auto assign address offset:**

      If the Gateway In is configured to be an AXI4-Lite interface, this option allows an address offset to be automatically assigned to the register within the AXI4-Lite interface that the Gateway In is mapped to.
    - **Address offset:** If Auto assign address offset is not checked, then this entry box allows you to explicitly specify an address offset to use. Must be a multiple of 4.
    - **Interface Name:** If the Gateway In is configured to be an AX4-Lite interface, assigns a unique name to this interface. This name can be used to differentiate between multiple AXI4-Lite interfaces in the design. When using the IP Catalog flow, you can expect to see an interface in the IP that Model Composer creates with the name <design_name>_<interface_name>_ s_axi.

      IMPORTANT! *The **Interface Name** must be composed of alphanumeric characters (lowercase alphabetic) or an underscore (_) only, and must begin with a lowercase alphabetic character. axi4_lite1 is acceptable, 1Axi4-Lite is not.*
    - **Description:** Additional designer comments about this Gateway In that is captured in the interface documentation.
  - **Constraints:**
    - **IOB Timing Constraint:** In hardware, a Gateway In is realized as a set of input/output buffers (IOBs). There are two constraint options: None, and Data Rate.

If None is selected, no timing constraints for the IOBs are put in the user constraint file produced by Model Composer. This means the paths from the IOBs to synchronous elements are not constrained.

If Data Rate is selected, the IOBs are constrained at the data rate at which the IOBs operate. The rate is determined by System Clock Period provided on the System Generator token and the sample rate of the Gateway relative to the other sample periods in the design.

- **Specify IOB location constraints:** Checking this option allows IOB location constraints and I/O standards to be specified.
- **IOB pad locations, e.g. {'MSB', ..., 'LSB'}:** IOB pin locations can be specified as a cell array of strings in this edit box. The locations are package-specific.
- **IO Standards, e.g. {'MSB', ..., 'LSB'}:** I/O standards can be specified as a cell array of strings in this edit box. The locations are package-specific.

Gateway Out

Xilinx Gateway Out blocks are the outputs from the HDL portion of your Simulink® design. This block converts the Model Composer fixed-point or floating-point data type into a Simulink integer, single, double or fixed-point data type.

According to its configuration, the Gateway Out block can either define an output port for the top level of the HDL design generated by Model Composer, or be used simply as a test point that is trimmed from the hardware representation

Gateway Blocks

As listed below, the Xilinx Gateway Out block is used to provide the following functions:

- Convert data from a Model Composer fixed-point or floating-point data type into a Simulink integer, single, double, or fixed-point data type.
- Define I/O ports for the top level of the HDL design generated by Model Composer. A Gateway Out block defines a top-level output port.
- Define test bench result vectors when the Model Composer **Create Testbench** box is checked. In this case, during HDL code generation, the outputs from the block that occur during Simulink simulation are logged as logic vectors in a data file. For each top level port, an HDL component is inserted in the top-level test bench that checks this vector against expected results during HDL simulation.
- Name the corresponding output port on the top-level HDL entity.

Block Parameters

- **Basic tab:** Parameters specific to the Basic tab are as follows.
    - **Propagate data type to output:** This option is useful when you instantiate a Model Composer design as a sub-system into a Simulink design. Instead of using a Simulink double as the output data type by default, the Model Composer data type is propagated to an appropriate Simulink data type according to the following table:

Table 55: **Propagate Data Type Output**

Model Composer Data Type	Simulink Data Type
XFloat_8_24	single
XFloat_11_53	double
Custom floating-point precision data type exponent width and fraction width less than those for single precision	single
Custom floating-point precision data type with exponent width or fraction width greater than that for single precision	double
XFix_<width>_<binpt>	sfix<width>_EN<binpt>
UFix_<width>_<binpt>	ufix<width>_EN<binpt>
XFix_<width>_0 where width is 8, 16 or 32	int<width> where width is 8, 16 or 32
UFix_<width>_0 where width is 8, 16 or 32	uint<width> where width is 8, 16 or 32
XFix_<width>_0 where width is other than 8, 16 or 32	sfix<width>
UFix_<width>_0 where width is other than 8, 16 or 32	ufix<width>

    - **Translate into Output Port:** Having this box unchecked prevents the gateway from becoming an actual output port when translated into hardware. This checkbox is on by default, enabling the output port. When this option is not selected, the Gateway Out block is used only during debugging, where its purpose is to communicate with Simulink Sink blocks for probing portions of the design. In this case, the Gateway Out block will turn gray in color, indicating that the gateway will not be translated into an output port.
- **Implementation tab:** Parameters specific to the Implementation tab are as follows.
    - **Interface Options:**
        - **Interface:**
            - **None:** During HDL Netlist generation, this Gateway Out will be translated as an Output Port at the top level.
            - **AXI4-Lite:** During HDL Netlist Generation, an AXI4-Lite interface will be created and the Gateway Out will be mapped to one of the registers within the AXI4-Lite interface.
            - **Interrupt:** During an IP Catalog Generation, this Gateway Out will be tagged as an Interrupt output port when the Model Composer design is packaged into an IP module that can be included in the Vivado® IP catalog.

- **Auto assign address offset:** If a Gateway Out is configured to be an AXI4-Lite interface, this option allows an address offset to be automatically assigned to the register within the AXI4-Lite interface that the Gateway Out is mapped to.
- **Address offset:** If Auto assign address offset is not checked, then this entry box allows you to explicitly specify a address offset to use. Must be a multiple of 4.
- **Interface Name:** If the Gateway Out is configured to be an AX4-Lite interface, assigns a unique name to this interface. This name can be used to differentiate between multiple AXI4-Lite interfaces in the design. When using the IP Catalog flow, you can expect to see an interface in the IP that Model Composer creates with the name <design_name>_<interface_name>_ s_axi.

  IMPORTANT! *The* ***Interface Name*** *must be composed of alphanumeric characters (lowercase alphabetic) or an underscore (_) only, and must begin with a lowercase alphabetic character. axi4_lite1 is acceptable, 1Axi4-Lite is not.*

- **Description:** Additional designer comments about this Gateway Out that is captured in the interface documentation.

- **Constraints:**
  - **IOB Timing Constraint:** In hardware, a Gateway Out is realized as a set of input/output buffers (IOBs). There are three ways to constrain the timing on IOBs. They are None, Data Rate, and Data Rate, Set 'FAST' Attribute.

    If **None** is selected, no timing constraints for the IOBs are put in the user constraint file produced by Model Composer. This means the paths from the IOBs to synchronous elements are not constrained.

    If **Data Rate** is selected, the IOBs are constrained at the data rate at which the IOBs operate. The rate is determined by System Clock Period provided on the System Generator token and the sample rate of the Gateway relative to the other sample periods in the design. For example, the following OFFSET = OUT constraints are generated for a Gateway Out named 'Dout' that is running at the system period of 10 ns:

```
# Offset out constraints
NET "Dout(0)" OFFSET = OUT : 10.0 : AFTER "clk";
NET "Dout(1)" OFFSET = OUT : 10.0 : AFTER "clk";
NET "Dout(2)" OFFSET = OUT : 10.0 : AFTER "clk";
```

If **Data Rate, Set 'FAST' Attribute** is selected, the OFFSET = OUT constraints described above are produced. In addition, a FAST slew rate attribute is generated for each IOB. This reduces delay but increases noise and power consumption. For the previous example, the following additional attributes are added to the constraints file.

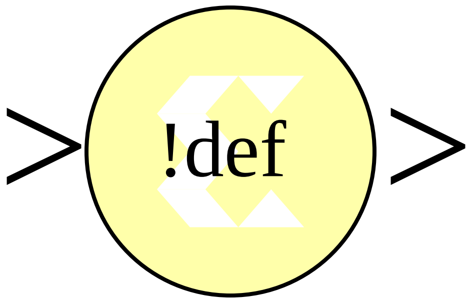
```
NET "Dout(0)" FAST;
NET "Dout(1)" FAST;
NET "Dout(2)" FAST;
```

- **Specify IOB Location Constraints:** Checking this option allows IOB location constraints to be specified.
- **IOB Pad Locations, e.g. {'MSB', ..., 'LSB'}:** IOB pin locations can be specified as a cell array of strings in this edit box. The locations are package-specific.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Indeterminate Probe

The output of the Xilinx Indeterminate Probe indicates whether the input data is indeterminate (MATLAB value NaN). An indeterminate data value corresponds to a VHDL indeterminate logic data value of 'X'.



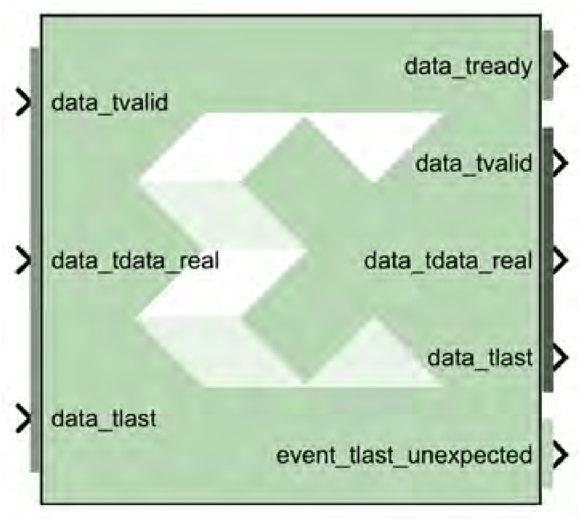
The probe accepts any Xilinx signal as input and produces a double signal as output. Indeterminate data on the probe input will result in an assertion of the output signal indicated by a value one. Otherwise, the probe output is zero.

Interleaver/De-interleaver 8.0

.

***Note*:** This block goes into the FPGA fabric and is a Licensed Core. Please visit the Xilinx web site to purchase the appropriate core license.

The Xilinx Interleaver Deinterleaver block implements an interleaver or a deinterleaver using an AXI4-compliant block interface. An interleaver is a device that rearranges the order of a sequence of input symbols. The term symbol is used to describe a collection of bits. In some applications, a symbol is a single bit. In others, a symbol is a bus.



The classic use of interleaving is to randomize the location of errors introduced in signal transmission. Interleaving spreads a burst of errors out so that error correction circuits have a better chance of correcting the data.

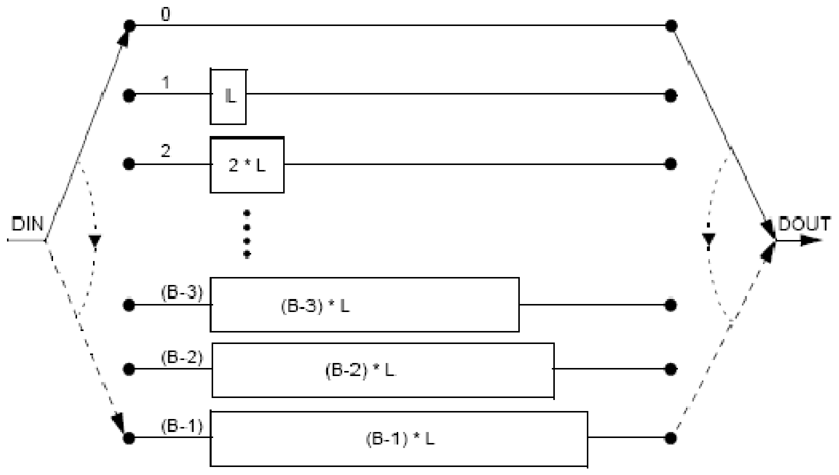
If a particular interleaver is used at the transmit end of a channel, the inverse of that interleaver must be used at the receive end to recover the original data. The inverse interleaver is referred to as a de-interleaver.

Two types of interleaver/de-interleavers can be generated with this LogiCORE™: Forney Convolutional and Rectangular Block. Although they both perform the general interleaving function of rearranging symbols, the way in which the symbols are rearranged and their methods of operation are entirely different. For very large interleavers, it might be preferable to store the data symbols in external memory. The core provides an option to store data symbols in internal FPGA RAM or in external RAM.

Forney Convolutional Operation

The figure below, shows the operation of a Forney Convolutional Interleaver. The core operates as a series of delay line shift registers. Input symbols are presented to the input commutator arm on DIN. Output symbols are extracted from the output commutator arm on DOUT. DIN and DOUT are fields in the AXI Data Input and Data Output channels, respectively. Output symbols are extracted from the output commutator arm on DOUT. Both commutator arms start at branch 0 and advance to the next branch after the next rising clock edge. After the last branch (B-1) has been reached, the commutator arms both rotate back to branch 0 and the process is repeated.

Figure 334: **Interleaver**

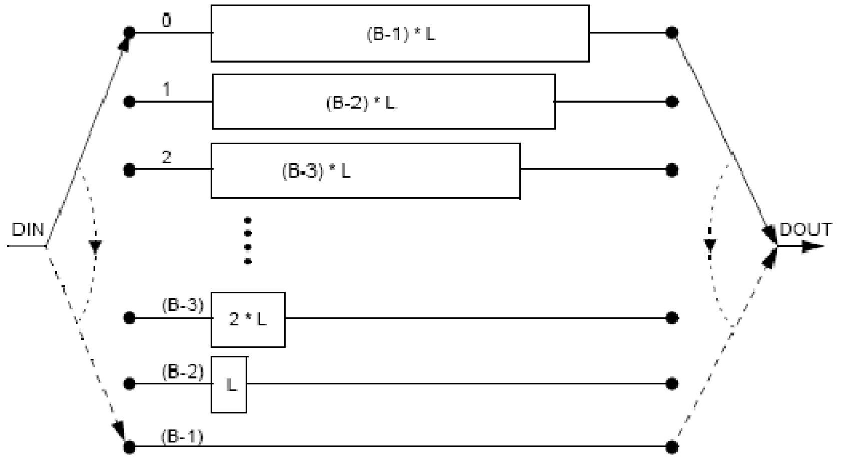


In the figure above, the branches increase in length by a uniform amount, L. The core allows interleavers to be specified in this way, or the branch lengths can be passed in using a file, allowing each branch to be any length.

Although branch 0 appears to be a zero-delay connection, there will still be a delay of a number of clock cycles between DIN and DOUT because of the fundamental latency of the core. For clarity, this is not illustrated in the figure.

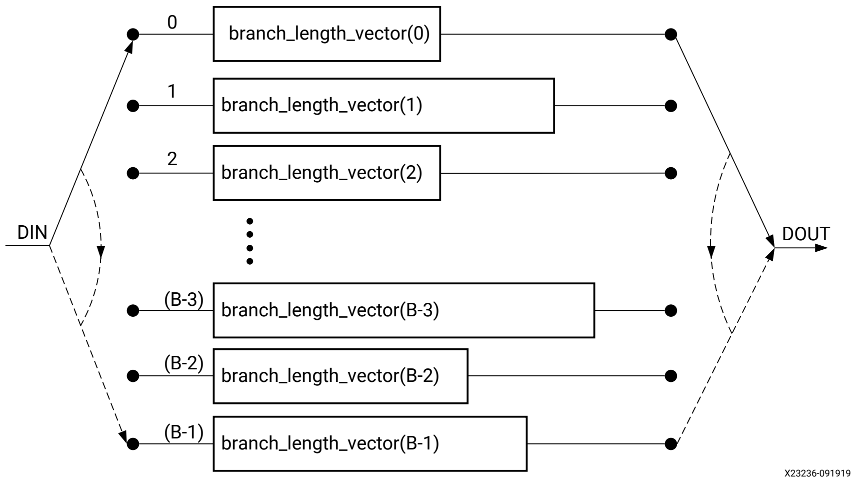
The only difference between an interleaver and a de-interleaver is that branch 0 is the longest in the de-interleaver and the branch length is decremented by L rather than incremented. Branch (B-1) has length 0. This is illustrated in the figure below:

Figure 335: **De-interleaver**



If a file is used to specify the branch lengths, as shown below, it is arbitrary whether the resulting core is called an interleaver or de-interleaver. All that matters is that one must be the inverse of the other. If a file is used, each branch length is individually controllable. This is illustrated in the figure below. For the file syntax, please consult the LogiCORE product specification.

Figure 336: **Interleaver/De-Interleaver**



The reset pin (aresetn) sets the commutator arms to branch 0, but does not clear the branches of data.

Configuration Swapping

It is possible for the core to store a number of pre-defined configurations. Each configuration can have a different number of branches and branch length constant. It is even possible for each configuration to have every individual branch length defined by file.

The configuration can be changed at any time by sending a new CONFIG_SEL value on the AXI Control Channel. This value takes effect when the next block starts. The core assumes all configurations are either for an interleaver or de-interleaver, depending on what was selected in the GUI. It is possible to switch between interleaving and de-interleaving by defining the individual branch lengths for every branch of each configuration. The details for each configuration are specified in a COE file.

For details, please consult the Configuration Swapping section of the *Interleaver/De-Interleaver LogiCORE IP Product Guide* (PG049).

Rectangular Block Operation

The Rectangular Block Interleaver works by writing the input data symbols into a rectangular memory array in a certain order and then reading them out in a different, mixed-up order. The input symbols must be grouped into blocks. Unlike the Convolutional Interleaver, where symbols can be continuously input, the Rectangular Block Interleaver inputs one block of symbols and then outputs that same block with the symbols rearranged. No new inputs can be accepted while the interleaved symbols from the previous block are being output.

The rectangular memory array is composed of a number of rows and columns as shown in the following figure.

Row\Column	0	1	...	(C-2)	(C-1)
0					
1					
..					
(R-2)					
(R-1)					

The Rectangular Block Interleaver operates as follows:

1. All the input symbols in an entire block are written row-wise, left to right, starting with the top row.
2. Inter-row permutations are performed if required.
3. Inter-column permutations are performed if required.
4. The entire block is read column-wise, top to bottom, starting with the left column.

The Rectangular Block De-interleaver operates in the reverse way:

1. All the input symbols in an entire block are written column-wise, top to bottom, starting with the left column.
2. Inter-row permutations are performed if required.
3. Inter-column permutations are performed if required.
4. The entire block is read row-wise, left to right, starting with the top row.

Refer to the *Interleaver/De-Interleaver LogiCORE IP Product Guide* (PG049) for examples and more detailed information on the Rectangular Block Interleaver.

AXI Interface

The AXI SID v7.1 has the following interfaces:

- A non AXI-channel interface for ACLK, ACLKEN and ARESETn
- A non AXI-channel interface for external memory (if enabled)

- A non AXI-channel interface for miscellaneous events
  - event_tlast_unexpected
  - event_tlast_missing (available only in Rectangular mode)
  - event_halted (optional, available when Master channel TREADY is enabled)
  - event_col_valid (optional)
  - event_col_sel_valid (optional)
  - event_row_valid (optional)
  - event_row_sel_valid (optional)
  - event_block_size_valid (optional)
- An AXI slave channel to receive configuration information (s_axis_ctrl) consisting of:
  - s_axis_ctrl_tvalid
  - s_axis_ctrl_tready
  - s_axis_ctrl_tdata

  The control channel is only enabled when the core is configured in such a way to require it.
- An AXI slave channel to receive the data to be interleaved (s_axis_data) consisting of:
  - s_axis_data_tvalid (This is the equivalent of ND pin of SID v6.0 block; No longer optional)
  - s_axis_data_tready
  - s_axis_data_tdata
  - s_axis_data_tlast
- An AXI master channel to send the data that has been interleaved (m_axis_data) consisting of:
  - m_axis_data_tvalid
  - m_axis_data_tready
  - m_axis_data_tdata
  - m_axis_data_tuser
  - m_axis_data_tlast

AXI Ports that are Unique to this Block

This HDL block exposes the AXI Control and Data channels as a group of separate ports based on the following sub-field names.

Note: Refer to the document LogiCORE IP Interleaver/De-interleaver v8.0 for an explanation of the bits in the specified sub-field name.

- **Control Channel Input Signals:**
  - **s_axis_ctrl_tdata_config_sel:**

    A sub-field port that represents the CONFIG_SEL field in the Control Channel vector. Available when in Forney mode and Number of configurations is greater than one.
  - **s_axis_ctrl_tdata_row:**

    A sub-field port that represents the ROW field in the Control Channel vector. Available when in Rectangular mode and Row type is Variable.
  - **s_axis_ctrl_tdata_row_sel:**

    A sub-field port that represents the ROW_SEL field in the Control Channel vector. Available when in Rectangular mode and Row type is Selectable.
  - **s_axis_ctrl_tdata_col:** A sub-field port that represents the COL field in the Control Channel vector. Available when in Rectangular mode and Column type is Variable.
  - **s_axis_ctrl_tdata_col_sel:**

    A sub-field port that represents the COL_SEL field in the Control Channel vector. Available when in Rectangular mode and Column type is Selectable.
  - **s_axis_ctrl_tdata_block_size:** A sub-field port that represents the COL field in the Control Channel vector. Available when in Rectangular mode and Block Size type is Variable.
- **DATA Channel Input Signals:**
  - **s_axis_data_tdata_din:**

    Represents the DIN field of the Input Data Channel.
- **DATA Channel Output Signals:**
  - **m_axis_data_tdata_dout:**

    Represents the DOUT field of the Output Data Channel.
- **TUSER Channel Output Signals:**
  - **m_axis_data_tuser_fdo:** Represents the FDO field of the Output TUSER Channel. Available when in Forney mode and Optional FDO pin has been selected on the GUI.
  - **m_axis_data_tuser_rdy:** Represents the RDY field of the Output TUSER Channel. Available when in Forney mode and Optional RDY pin has been selected on the GUI.

- **m_axis_data_tuser_block_start:** Represents the BLOCK_START field of the Output TUSER Channel. Available when in Rectangular mode and Optional BLOCK_START pin has been selected on the GUI.
- **m_axis_data_tuser_block_end:** Represents the BLOCK_END field of the Output TUSER Channel. Available when in Rectangular mode and Optional BLOCK_END pin has been selected on the GUI.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

- **Basic tab:** Parameters specific to the Basic Parameters tab are as follows.
    - **Memory Style:** Select **Distributed** if all the Block Memories are required elsewhere in the design; select **Block** to use Block Memory where ever possible; select **Automatic** and let Model Composer use the most appropriate style of memory for each case, based on the required memory depth.
    - **Symbol Width:** This is the number of bits in the symbols to be processed.
    - **Type:** Select **Forney Convolutional** or **Rectangular Block**.
    - **Mode:** Select **Interleaver** or **Deinterleaver**
    - **Symbol memory:** Specifies whether or not the data symbols are stored in Internal FPGA RAM or in External RAM.
- **Forney tab:** Parameters specific to the Forney Parameters tab are as follows.
    - **Dimensions:**
        - **Number of branches:** 1 to 256 (inclusive)
    - **Architecture:**
        - **ROM-based:** Look-up table ROMs are used to compute some of the internal results in the block.
        - **Logic-based:** Logic circuits are used to compute some of the internal results in the block.

      Which option is best depends on the other core parameters. You should try both options to determine the best results. This parameter has no effect on the block behavior.
    - **Configurations:**
        - **Number of configurations:** If greater than 1, the block is generated with CONFIG_SEL and NEW_CONFIG inputs. The parameters for each configuration are defined in a COE file. The number of parameters defined must exactly match the number of configurations specified.

- **Length of Branches:** Branch length descriptions for Forney SID.
    - **constant_difference_between_consecutive_branches:** Specified by the Value parameter.
    - **use_coe_file_to_define_branch_lengths:** Location of file is specified by the COE File parameter.
    - **coe_file_defines_individual_branch_lengths_for_every_branch_in_each_configuration:** Location of file is specified by the COE File parameter.
    - **coe_file_defines_branch_length_constant_for_each_configuration:** Location of file is specified by the COE File parameter.
    - **Value:** 1 to MAX (inclusive). MAX depends on the number of branches and size of block input. Branch length must be an array of either length one or number of branches. If the array size is one, the value is used as a constant difference between consecutive branches. Otherwise, each branch has a unique length.
    - **COE File:** The branch lengths are specified from a file

- **Rectangular Parameters #1 Tab:** Parameters specific to the Rectangular Parameters #1 tab are as follows.
    - **Number of Rows:**
        - **Value:** This parameter is relevant only when the Constant row type is selected. The number of rows is fixed at this value.
        - **Row Port Width:** This parameter is relevant only when the Variable row type is selected. It sets the width of the ROW input bus. The smallest possible value should be used to keep the underlying LogiCORE as small as possible.
        - **Minimum Number of Rows:** This parameter is relevant only when the Variable row type is selected. In this case, the core has to potentially cope with a wide range of possible values for the number of rows. If the smallest value that will actually occur is known, then the amount of logic in the LogiCORE can sometimes be reduced. The largest possible value should be used for this parameter to keep the core as small as possible.
        - **Number of Values:** This parameter is relevant only when you select the **Selectable** row type. This parameter defines how many valid selection values have been defined in the COE file. You should only add the number of select values you need.
        - **Row Type:**
            - **Constant:** The number of rows is always equal to the Row Constant Value parameter.
            - **Variable:** The number of rows is sampled from the ROW input at the start of each new block. Row permutations are not supported for the variable row type.
            - **Selectable:** ROW_SEL is sampled at the start of each new block. This value is then used to select from one of the possible values for the number of rows provided in the COE file.

- **Number of Columns:**
  - **Value:** This parameter is relevant only when you select the **Constant** column type is selected. The number of columns is fixed at this value.
  - **COL Port Width:** This parameter is relevant only when you select the Variable column type. It sets the width of the COL input bus. The smallest possible value should be used to keep the underlying LogiCORE™ as small as possible.
  - **Minimum Number of Columns:** This parameter is relevant only when you select the **Variable** column type is selected. In this case, the core has to potentially cope with a wide range of possible values for the number of columns. If the smallest value that will actually occur is known, then the amount of logic in the LogiCORE can sometimes be reduced. The largest possible value should be used for this parameter to keep the core as small as possible.
  - **Number of Values:** This parameter is relevant only when you select the **Selectable** column type. This parameter defines how many valid selection values have been defined in the COE file. You should only add the number of select values you need.
  - **Column Type:**
    - **Constant:** The number of columns is always equal to the Column Constant Value parameter.
    - **Variable:** The number of columns is sampled from the `COL` input at the start of each new block. Column permutations are not supported for the variable column type.
    - **Selectable:** COL_SEL is sampled at the start of each new block. This value is then used to select from one of the possible values for the number of columns provided in the COE file.

- **Rectangular Parameters #2 Tab:** Parameters specific to the Rectangular Parameters #2 tab are as follows.
  - **Permutations Configuration:**
    - **Row permutations:**
      - **None:** This tells Model Composer that row permutations are not to be performed.
      - **Use COE file:** This tells Model Composer that a row permute vector exists in the COE file, and that row permutations are to be performed. Remember this is possible only for un-pruned interleaver/deinterleavers.
    - **Column permutations:**
      - **None:** This tells Model Composer that column permutations are not to be performed
      - **Use COE file:** This tells Model Composer that a column permute vector exists in the `COE` file, and that column permutations are to be performed. Remember this is possible only for un-pruned interleaver/deinterleavers.

        - **COE File:** Specify the pathname to the `COE` file.
    - **Block Size:**
        - **Value:** This parameter is relevant only when you select the **Constant** block size type. The block size is fixed at this value.
        - **BLOCK_SIZE Port Width:** This parameter is relevant only if the Variable block size type is selected. It sets the width of the BLOCK_SIZE input bus. The smallest possible value should be used to keep the core as small as possible.
        - **Block Size Type:**
            - **Constant:** The block size never changes. The block can be pruned (block size < row * col). The block size must be chosen so that the last symbol is on the last row. An un-pruned interleaver will use a smaller quantity of FPGA resources than a pruned one, so pruning should be used only if necessary.
            - **Rows*Columns:**

              If the number of rows and columns is constant, selecting this option has the same effect as setting the block size type to constant and entering a value of rows * columns for the block size.

              If the number of rows or columns is not constant, selecting this option means the core will calculate the block size automatically whenever a new row or column value is sampled. Pruning is impossible with this block size type.
            - **Variable:**

              Block size is sampled from the BLOCK_SIZE input at the beginning of every block. The value sampled on BLOCK_SIZE must be such that the last symbol falls on the last row, as previously described.

              If the block size is already available external to the core, selecting this option is usually more efficient than selecting "rows * columns" for the block size type. Row and column permutations are not supported for the Variable block size type.
- **Port Parameters #1 tab:** Parameters specific to the Port Parameters tab are as follows.
    - **Control Signals:**
        - **ACLKEN:** When ACLKEN is de-asserted (Low), all the synchronous inputs are ignored and the block remains in its current state.
        - **ARESETn (Active-Low).:** The Active-Low synchronous clear input always takes priority over ACLKEN.
    - **Status Signals:**

        - **COL_VALID:** This optional output is available when a variable number of columns is selected. If an illegal value is sampled on the `s_axis_ctrl_tdata_col` input, `event_col_valid` will go Low a predefined number of clock cycles later.
        - **COL_SEL_VALID:** This optional output (`event_col_sel_valid`) is available when a selectable number of columns is chosen. The event pins are `event_col_valid`, `event_col_sel_valid`, `event_row_valid`, `event_row_sel_valid`, `event_block_size_valid` (in the same order as in the options on the GUI).
        - **ROW_VALID:** This optional output is available when a selectable number of rows is chosen.
        - **ROW_SEL_VALID:** This optional output is available when a selectable number of rows is chosen.
        - **BLOCK_SIZE_VALID:** This optional output is available when the block size is not constant, that is, if the block size type is either Variable or equal to Rows * Columns.
- **Port Parameters #2 tab:** Parameters specific to the Port Parameters #2 tab are as follows.
    - **Data Output Channel Options:**
        - **TREADY:** TREADY for the Data Input Channel. Used by the Symbol Interleaver/De-interleaver to signal that it is ready to accept data.
        - **FDO:** Adds a data_tuser_fdo (First Data Out) output port.
        - **RDY:** Adds a data_tuser_rdy output port.
        - **BLOCK_START:** Adds a data_tuser_block_start output port.
        - **BLOCK_END:** Adds a data_tuser_block_end output port.
    - **Pipelining:**
        - **Pipelining:** Pipelines the underlying LogiCORE for Minimum, Medium, or Maximum performance.

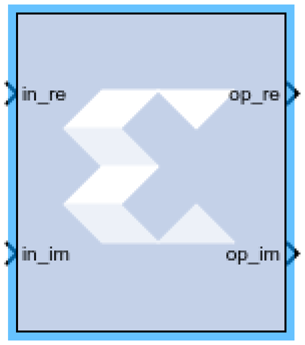
Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP Interleaver/De-interleaver v8.0

Inverse FFT

The Xilinx Inverse FFT block performs a fast inverse (or backward) Fourier transform (IDFT), which undoes the process of Discrete Fourier Transform (DFT). The Inverse FFT maps the signal back from the frequency domain into the time domain.



The IDFT of a sequence $\{F_n\}$ can be defined as:

$$x(n)=\frac{1}{N}\sum_{k=0}^{N-1} X[k]e^{+j\frac{2\pi}{N}nk} \quad for \quad n=0,1,2,\ldots,N-1$$

where N is the transform length, k is used to denote the frequency domain ordinal, and n is used to represent the time-domain ordinal.

The Inverse FFT (IFFT) is computed by conjugating the phase factors of the corresponding forward FFT.

The Inverse FFT block is ideal for implementing simple inverse Fourier transforms. If your Inverse FFT implementation will use more complicated transform features such as an AXI4-Stream-compliant interface, a real time throttle scheme, Radix-4 Burst I/O, or Radix-2 Lite Burst I/O, use the Xilinx Fast Fourier Transform 9.1 block in your design instead of the Inverse FFT block.

In the Vivado® design flow, the Inverse FFT block is inferred as "LogiCORE™ IP Fast Fourier Transform v9.1" for code generation. Refer to the document LogiCORE IP Fast Fourier Transform v9.1 for details on this LogicCore IP.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

Parameters specific to the Xilinx Inverse FFT block are:

- **Transform Length:** Select the desired point size ranging from 8 to 65536.

- **Scale Result by FFT length:** If selected, data is scaled between IFFT stages using a scaling schedule determined by the Transform Length setting. If not selected, data is unscaled, and all integer bit growth is carried to the output.
- **Natural Order:** If selected, the output of the Inverse FFT block will be ordered in natural order. If not selected, the output of the Inverse FFT block will be ordered in bit/digit reversed order.
- **Optimize for:** Directs the block to be optimized for either speed (Performance) or area (Resources) in the generated hardware.

  Note: If you selected **Resources** and the input sample period is 8 times slower than the system sample period, the block implements Radix-2 Burst I/O architecture. Otherwise, Pipeline Streaming I/O architecture is used.
- **Optional Port:**
  - **Provide start frame port:** Adds `start_frame_in` and `start_frame_out` ports to the block. The signals on these ports can be used to synchronize frames at the input and output of the Inverse FFT block. See Adding Start Frame Ports to Synchronize Frames for a description of the operation of these two ports.

Context Based Pipeline vs. Radix Implementation

Pipelined Streaming I/O and Radix-2 Burst I/O architectures are supported by the Inverse FFT block. Radix-4 Burst I/O architecture is implemented when the **Optimize for: Resources** block parameter is selected and the sample rate of the inputs is 8 times slower than the system rate. In all other configurations Pipelined Streaming I/O architecture is implemented by default.

Input Data Type Support

The Inverse FFT block accepts inputs of varying bit widths with changeable binary point location, such as Fix_16_0 or Fix_30_10, etc. in unscaled block configuration. For the scaled configuration, the input is supported in the same format as the Fast Fourier Transform 9.1 block. The Fast Fourier Transform 9.1 block accepts input values only in the normalized form in the format of Fix_x_[x-1] (for example, Fix_16_15), so the inputs are 2's complement with a single sign/integer bit.

Latency Value Displayed on the Block

The latency value depends on parameters selected by the user, and the corresponding latency value is displayed on the Inverse FFT block icon in the Simulink model.

Automatic Fixed Point and Floating Point Support

Signed fixed point and floating point data types are supported.

For floating point input, either scaled or unscaled data can be selected in the Inverse FFT block parameters. In the Fast Fourier Transform 9.1 block, the floating point data type is accepted only when the scaled configuration is selected by the user.

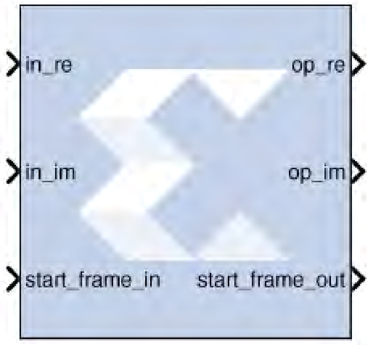
Handling Overflow for Scaled Configuration

The Inverse FFT block uses a conservative schedule to avoid overflow scenarios. This schedule sets the scaling value for the corresponding FFT stages in a way that makes sure no overflow occurs.

Adding Start Frame Ports to Synchronize Frames

Selecting **Provide start frame port** in the Inverse FFT block properties dialog box adds `start_frame_in` and `start_frame_out` ports at the input and output of the Inverse FFT block. These ports are used to synchronize frames at the input and output of the Inverse FFT block.

Figure 337: **Adding Start Frame Ports**



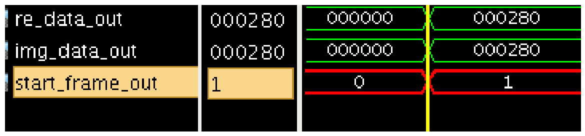
You must provide a valid input at the `start_frame_in` port. When the `start_frame_in` signal is asserted, an impulse is generated at the start of every frame to signal the Inverse FFT block to start processing the frame. The frame size is the Transform Length entered in the block parameters dialog box.

The `start_frame_out` port provides the information as to when the output frames start. An impulse at the start of every frame on the output side helps in tracking the block behavior.

The Inverse FFT block has a frame alignment requirement and these ports help the block operate in accordance with this requirement.

The figure below shows that as soon as the output is processed by the Inverse FFT block the `start_frame_out` signal becomes High (1).

Figure 338: **Output**



The following apply to the **Provide start frame port** option and the start frame ports added to the FFT block when the option is enabled:

- The **Provide start frame port** option selection is valid only for Pipelined Streaming I/O architecture. See Context Based Pipeline vs. Radix Implementation for a description of the conditions under which Pipelined Streaming I/O architecture is implemented.
- The option is valid only for input of type fixed point.
- Verilog is supported for netlist generation currently, when the **Provide start frame port** option is selected.

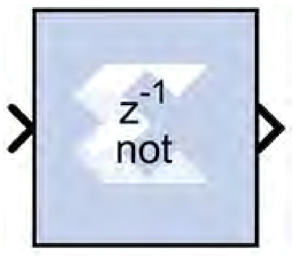
***Note*:** The first sample input to the Inverse FFT block may be ignored and users are advised to drive the input data accordingly.

LogiCORE Documentation

LogiCORE IP Fast Fourier Transform v9.1

Inverter

The Xilinx Inverter block calculates the bitwise logical complement of a fixed-point number. The block is implemented as a synthesizable VHDL module.



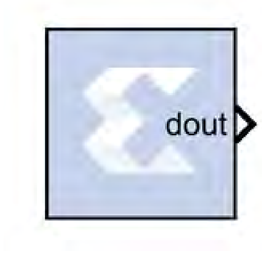
Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LFSR

The Xilinx LFSR block implements a Linear Feedback Shift Register (LFSR). This block supports both the Galois and Fibonacci structures using either the XOR or XNOR gate and allows a re-loadable input to change the current value of the register at any time. The LFSR output and re-loadable input can be configured as either serial or parallel ports



Block Interface

Table 56: **Block Interface**

Port Name	Port Description	Port Type
din	Data input for re-loadable seed	Optional serial or parallel input
load	Load signal for din	Optional boolean input
rst	Reset signal	Optional boolean input
en	Enable signal	Optional boolean input
dout	Data output of LFSR	Required serial or parallel output

As shown in the table above, there can be between 0 and 4 block input ports and exactly one output port. If the configuration selected requires 0 inputs, the LFSR is set up to start at a specified initial seed value and will step through a repeatable sequence of states determined by the LFSR structure type, gate type, and initial seed.

The optional din and load ports provide the ability to change the current value of the LFSR at runtime. After the load completes, the LFSR behaves as with the 0 input case, and starts up a new sequence based upon the newly loaded seed, and the statically configured LFSR options for structure and gate type.

The optional rst port will reload the statically specified initial seed of the LFSR and continue on as before after the rst signal goes low. And when the optional en port goes low, the LFSR will remain at its current value with no change until the en port goes high again.

Block Parameters

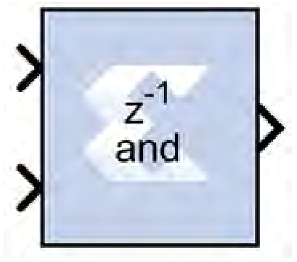
The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:** Parameters specific to the Basic tab are as follows:
  - **Type:** Fibonacci or Galois. This field specifies the structure of the feedback. Fibonacci has one XOR (or XNOR) gate at the beginning of the register chain that XORs (or XNORs) the taps together with the result going into the first register. Galois has one XOR(or XNOR) gate for each tap and gates the last register in the chains output with the input to the register at that tap.
  - **Gate type:** XOR or XNOR. This field specifies the gate used by the feedback signals.
  - **Number of bits in LFSR:** This field specifies the number of registers in the LFSR chain. As a result, this number specifies the size of the input and output when selected to be parallel.
  - **Feedback polynomial:** This field specifies the tap points of the feedback chain and the value must be entered in hex with single quotes. The lsb of this polynomial always must be set to 1 and the msb is an implied 1 and is not specified in the hex input. Please see the Xilinx application note titled Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators for more information on how to specify this equation and for optimal settings for the maximum repeating sequence.
  - **Initial value:** This field specifies the initial seed value where the LFSR begins its repeating sequence. The initial value might not be all zeroes when choosing the XOR gate type and might not be all ones when choosing XNOR, as those values will stall the LFSR.
- **Advanced tab:** Parameters specific to the Advanced tab are as follows:
  - **Use reloadable seed value :** This field specifies whether or not an input is needed to reload a dynamic LFSR seed value at run time.
  - **Parallel input:** This field specifies whether the reloadable input seed is shifted in one bit at a time or if it happens in parallel.
  - **Parallel output:** This field specifies whether all of the bits in the LFSR chain are connected to the output or just the last register in the chain (serial or parallel).

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Logical

The Xilinx Logical block performs bitwise logical operations on fixed-point numbers. Operands are zero padded and sign extended as necessary to make binary point positions coincide; then the logical operation is performed and the result is delivered at the output port.



In hardware this block is implemented as synthesizable VHDL. If you build a tree of logical gates, this synthesizable implementation is best as it facilitates logic collapsing in synthesis and mapping.

Block Parameters

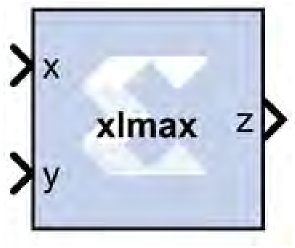
The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:** Parameters specific to the Basic tab are as follows:
    - **Logical function:** Specifies one of the following bitwise logical operators: AND, NAND, OR, NOR, XOR, XNOR.
    - **Number of inputs:** Specifies the number of inputs (1 - 1024).
    - **Logical Reduction Operation:** When the number of inputs is specified as 1, a unary logical reduction operation performs a bit-wise operation on the single operand to produce a single bit result. The first step of the operation applies the logical operator between the least significant bit of the operand and the next most significant bit. The second and subsequent steps apply the operator between the one-bit result of the prior step and the next bit of the operand using the same logical operator. The logical reduction operator implements the same functionality as that of the logical reduction operation in HDLs. The output of the logical reduction operation is always Boolean.
- **Output Type tab:** Parameters specific to the Output Type tab are as follows.
    - **Align binary point:** Specifies that the block must align binary points automatically. If not selected, all inputs must have the same binary point position.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

MCode

The Xilinx MCode block is a container for executing a user-supplied MATLAB function within Simulink. A parameter on the block specifies the M-function name. The block executes the M-code to calculate block outputs during a Simulink simulation. The same code is translated in a straightforward way into equivalent behavioral VHDL/Verilog when hardware is generated.



The block's Simulink® interface is derived from the MATLAB function signature, and from block mask parameters. There is one input port for each parameter to the function, and one output port for each value the function returns. Port names and ordering correspond to the names and ordering of parameters and return values.

The MCode block supports a limited subset of the MATLAB language that is useful for implementing arithmetic functions, finite state machines, and control logic.

The MCode block has the following three primary coding guidelines that must be followed:

- All block inputs and outputs must be of Xilinx fixed-point type.
- The block must have at least one output port.
- The code for the block must exist on the MATLAB path or in the same directory as the directory as the model that uses the block.

The example described below consists of a function `xlmax` which returns the maximum of its inputs. The second illustrates how to do simple arithmetic. The third shows how to build a finite state machine.

Configuring an MCode Block

The **MATLAB Function** parameter of an MCode block specifies the name of the block's M- code function. This function must exist in one of the three locations at the time this parameter is set. The three possible locations are:

- The directory where the model file is located.
- A subdirectory of the model directory named private.
- A directory in the MATLAB path.

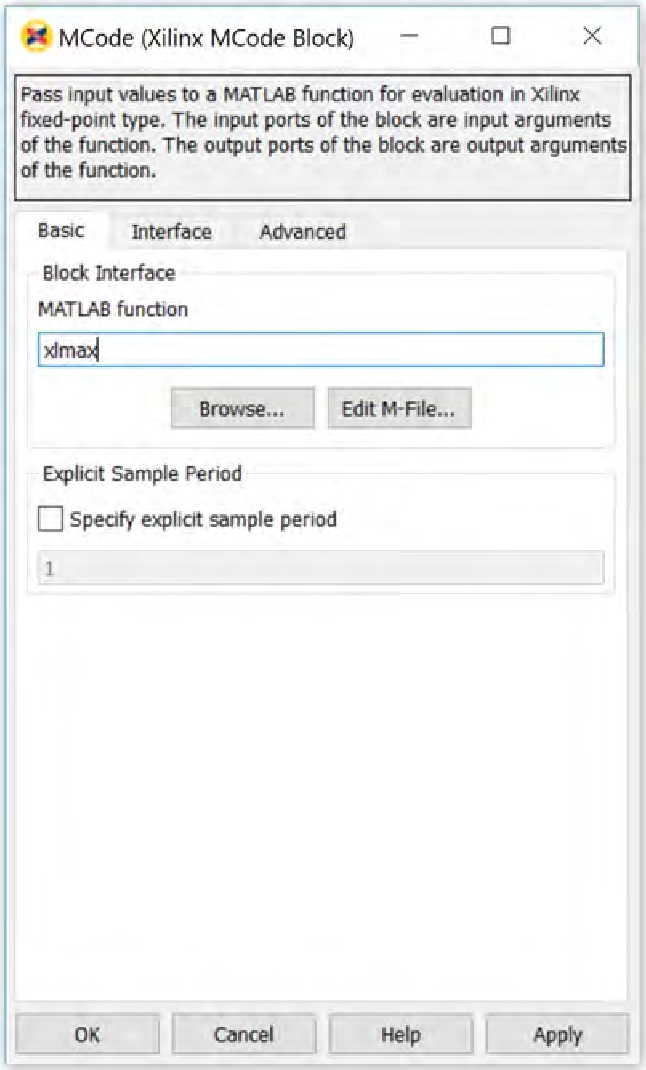
The block icon displays the name of the M-function. To illustrate these ideas, consider the file `xlmax.m` containing function `xlmax`:

```
function z = xlmax(x, y)
  if x > y
    z = x;
  else
  z = y;
  end
```

An MCode block based on the function `xlmax` will have input ports `x` and `y` and output port `z`.

The following figure shows how to set up an MCode block to use function `xlmax`.

Figure 339: **xlmax function**



MCode (Xilinx MCode Block)

Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

Block Interface

MATLAB function: xlmax

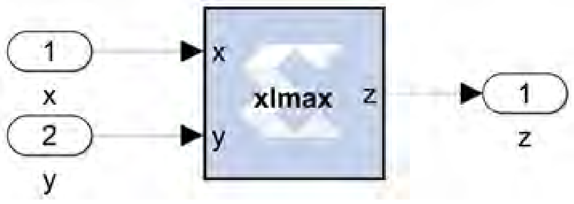
Explicit Sample Period

Specify explicit sample period [ ]

1

Once the model is compiled, the xlmax MCode block will appear like the block illustrated below.

Figure 340: **xlmax MCode block**



MATLAB Language Support

The MCode block supports the following MATLAB language constructs:

- Assignment statements
- Simple and compound `if/else/elseif` end statements
- `switch` statements
- Arithmetic expressions involving only addition and subtraction
- Addition
- Subtraction
- Multiplication
- Division by a power of two
- Relational operators:

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
~=	Not equal to

- Logical operators:

&	And
|	Or
~	Not

The MCode block supports the following MATLAB functions.

- Type conversion. The only supported data type is `xfix`, the Xilinx fixed-point type. The `xfix()` type conversion function is used to convert to this type. The conversion is done implicitly for integers but must be done explicitly for floating point constants. All values must be scalar; arrays are not supported.
- Functions that return `xfix` properties:

`xl_nbits()`	Returns number of bits
`xl_binpt()`	Returns binary point position
`xl_arith()`	Returns arithmetic type

- Bit-wise logical functions:

`xl_and()`	Bit-wise and

`xl_or()`	Bit-wise or
`xl_xor()`	Bit-wise xor
`xl_not()`	Bit-wise not

- Shift functions: `xl_lsh()` and `xl_rsh()`
- Slice function: `xl_slice()`
- Concatenate function: `xl_concat()`
- Reinterpret function: `xl_force()`
- Internal state variables: `xl_state()`
- MATLAB Functions:

`disp()`	Displays variable values
`error()`	Displays message and abort function
`isnan()`	Tests whether a number is NaN
`NaN()`	Returns Not-a-Number
`num2str()`	Converts a number to string
`ones(1,N)`	Returns 1-by-N vector of ones
`pi()`	Returns pi
`zeros(1,N)`	Returns 1-by-N vector of zeros

- **Data Types :**

  There are three kinds of `xfix` data types: unsigned fixed-point (`xlUnsigned`), signed fixed-point(`xlSigned`), and boolean (`xlBoolean`). Arithmetic operations on these data types produce signed and unsigned fixed-point values. Relational operators produce a boolean result. Relational operands can be any `xfix` type, provided the mixture of types makes sense. Boolean variables can be compared to boolean variables, but not to fixed-point numbers; boolean variables are incompatible with arithmetic operators. Logical operators can only be applied to boolean variables. Every operation is performed in full precision, for example, with the minimum precision needed to guarantee that no information is lost.

  Literal Constants

  Integer, floating-point, and boolean literals are supported. Integer literals are automatically converted to `xfix` values of appropriate width having a binary point position at zero. Floating-point literals must be converted to the `xfix` type explicitly with the `xfix()` conversion function. The predefined MATLAB values `true` and `false` are automatically converted to boolean literals.

  Assignment

  The left-hand side of an assignment can only contain one variable. A variable can be assigned more than once.

Control Flow

The conditional expression of an `if` statement must evaluate to a boolean. Switch statements can contain a `case` clause and an `otherwise` clause. The types of a switch selector and its cases must be compatible; thus, the selector can be boolean provided its cases are. All cases in a `switch` must be constant; equivalently, no `case` can depend on an input value.

When the same variable is assigned in several branches of a control statement, the types being assigned must be compatible. For example,

```
if (u > v)
  x = a;
else
  x = b;
end
```

is acceptable only if `a` and `b` are both boolean or both arithmetic.

- **Constant Expressions:**

  An expression is constant provided its value does not depend on the value of any input argument. Thus, for example, the variable `c` defined by

```
a = 1;
b = a + 2;
c = xfix({xlSigned, 10, 2}, b + 3.345);
```

can be used in any context that demands a constant.

xfix() Conversion

The `xfix()` conversion function converts a `double` to an `xfix`, or changes one `xfix` into another having different characteristics. A call on the conversion function looks like the following

```
x = xfix(type_spec, value)
```

Here `x` is the variable that receives the `xfix`. *type_spec* is a cell array that specifies the type of `xfix` to create, and value is the value being operated on. The `value` can be floating point or `xfix` type. The *type_spec* cell array is defined using curly braces in the usual MATLAB method. For example,

```
xfix({xlSigned, 20, 16, xlRound, xlWrap}, 3.1415926)
```

returns an `xfix` approximation to `pi`. The approximation is signed, occupies 20 bits (16 fractional), quantizes by rounding, and wraps on overflow.

The *type_spec* consists of 1, 3, or 5 elements. Some elements can be omitted. When elements are omitted, default element settings are used. The elements specify the following properties (in the order presented): `data type`, `width`, `binary point position`, `quantization mode`, and `overflow mode`. The `data type` can be `xlBoolean`, `xlUnsigned`, or `xlSigned`. When the type is `xlBoolean`, additional elements are not needed (and must not be supplied). For other types, `width` and `binary point position` must be supplied. The `quantization` and `overflow modes` are optional, but when one is specified, the other must be as well. Three values are possible for quantization: `xlTruncate`, `xlRound`, and `xlRoundBanker`. The default is `xlTruncate`. Similarly, three values are possible for overflow: `xlWrap`, `xlSaturate`, and `xlThrowOverflow`. For `xlThrowOverflow`, if an overflow occurs during simulation, an exception occurs.

All values in a *type_spec* must be known at compilation time; equivalently, no *type_spec* value can depend on an input to the function.

The following is a more elaborate example of an `xfix()` conversion:

```
width = 10, binpt = 4;
z = xfix({xlUnsigned, width, binpt}, x + y);
```

This assignment to `x` is the result of converting `x + y` to an unsigned fixed-point number that is 10 bits wide with 4 fractional bits using `xlTruncate` for quantization and `xlWrap` for overflow.

If several `xfix()` calls need the same *type_spec* value, you can assign the *type_spec* to a variable, then use the variable for `xfix()` calls. For example, the following is allowed:

```
proto = {xlSigned, 10, 4};
x = xfix(proto, a);
y = xfix(proto, b);
```

- **xfix Properties: xl_arith, xl_nbits, and xl_binpt:**

  Each `xfix` number has three properties: the arithmetic type, the bit width, and the binary point position. The MCode blocks provide three functions to get these properties of a fixed-point number. The results of these functions are constants and are evaluated when Simulink compiles the model.

  Function `a = xl_arith(x)` returns the arithmetic type of the input number `x`. The return value is either `1`, `2`, or `3` for `xlUnsigned`, `xlSigned`, or `xlBoolean` respectively.

  Function `n = xl_nbits(x)` returns the width of the input number `x`.

  Function `b = xl_binpt(x)` returns the binary point position of the input number `x`.

- **Bit-wise Operators: xl_or, xl_and, xl_xor, and xl_not :**

The MCode block provides four built-in functions for bit-wise logical operations: `xl_or`, `xl_and`, `xl_xor`, and `xl_not`.

Function `xl_or`, `xl_and`, and `xl_xor` perform bit-wise logical or, and, and xor operations respectively. Each function is in the form of

```
x = xl_op(a, b, ).
```

Each function takes at least two fixed-point numbers and returns a fixed-point number. All the input arguments are aligned at the binary point position.

Function `xl_not` performs a bit-wise logical not operation. It is in the form of `x = xl_not(a)`. It only takes one `xfix` number as its input argument and returns a fixed- point number.

The following are some examples of these function calls:

```
X = xl_and(a, b);
Y = xl_or(a, b, c);
Z = xl_xor(a, b, c, d);
N = xl_not(x);
```

- **Shift Operators: xl_rsh, and xl_lsh:**

  Functions `xl_lsh` and `xl_rsh` allow you to shift a sequence of bits of a fixed-point number. The function is in the form:

  `x = xl_lsh(a, n)` and `x = xl_rsh(a, n)` where `a` is a `xfix` value and `n` is the number of bits to shift.

  Left or right shift the fixed-point number by `n` number of bits. The right shift (`xl_rsh`) moves the fixed-point number toward the least significant bit. The left shift (`xl_lsh`) function moves the fixed-point number toward the most significant bit. Both shift functions are a full precision shift. No bits are discarded and the precision of the output is adjusted as needed to accommodate the shifted position of the binary point.

  Here are some examples:

```
% left shift a 5 bits
a = xfix({xlSigned, 20, 16, xlRound, xlWrap}, 3.1415926)
b = xl_rsh(a, 5);
```

  The output `b` is of type `xlSigned` with 21 bits and the binary point located at bit 21.

- **Slice Function: xl_slice:**

Function `xl_slice` allows you to access a sequence of bits of a fixed-point number. The function is in the form:

```
x = xl_slice(a, from_bit, to_bit).
```

Each bit of a fixed-point number is consecutively indexed from zero for the LSB up to the MSB. For example, given an 8-bit wide number with binary point position at zero, the LSB is indexed as 0 and the MSB is indexed as 7. The block will throw an error if the `from_bit` or `to_bit` arguments are out of the bit index range of the input number. The result of the function call is an unsigned fixed-point number with zero binary point position.

Here are some examples:

```
% slice 7 bits from bit 10 to bit 4
b = xl_slice(a, 10, 4);
% to get MSB
c = xl_slice(a, xl_nbits(a)-1, xl_nbits(a)-1);
```

- **Concatenate Function: xl_concat :**

  Function `x = xl_concat(hi, mid, ..., low)` concatenates two or more fixed-point numbers to form a single fixed-point number. The first input argument occupies the most significant bits, and the last input argument occupies the least significant bits. The output is an unsigned fixed-point number with binary point position at zero.

- **Reinterpret Function: xl_force:**

  Function `x = xl_force(a, arith, binpt)` forces the output to a new type with `arith` as its new arithmetic type and `binpt` as its new binary point position. The `arith` argument can be one of `xlUnsigned`, `xlSigned`, or `xlBoolean`. The `binpt` argument must be from 0 to the bit width inclusively. Otherwise, the block will throw an error.

- **State Variables: xl_state:**

An MCode block can have internal state variables that hold their values from one simulation step to the next. A state variable is declared with the MATLAB keyword persistent and must be initially assigned with an `xl_state` function call.

The following code models a 4-bit accumulator:

```
function q = accum(din, rst)
  init = 0;

  persistent s, s = xl_state(init, {xlSigned, 4, 0});
  q = s;
  if rst
    s = init;
 else
    s = s + din;
 end
```

The state variable `s` is declared as persistent, and the first assignment to `s` is the result of the `xl_state` invocation. The `xl_state` function takes two arguments. The first is the initial value and must be a constant. The second is the precision of the state variable. It can be a type cell array as described in the `xfix` function call. It can also be an `xfix` number. In the above code, if `s = xl_state(init, din)`, then state variable s will use `din` as the precision. The `xl_state` function must be assigned to a persistent variable.

The `xl_state` function behaves in the following way:

1. In the first cycle of simulation, the `xl_state` function initializes the state variable with the specified precision.
2. In the following cycles of simulation, the `xl_state` function retrieves the state value left from the last clock cycle and assigns the value to the corresponding variable with the specified precision.

`v = xl_state(init, precision)` returns the value of a state variable. The first input argument `init` is the initial value, the second argument `precision` is the precision for this state variable. The argument `precision` can be a cell arrary in the form of `{type, nbits, binpt}` or `{type, nbits, binpt, quantization,overflow}`. The `precision` argument can also be an `xfix` number.

`v = xl_state(init, precision, maxlen)` returns a vector object. The vector is initialized with `init` and will have `maxlen` for the maximum length it can be. The vector is initialized with `init`. For example, `v = xl_state(zeros(1, 8), prec, 8)` creates a vector of 8 zeros, `v = xl_state([], prec, 8)` creates an empty vector with 8 as maximum length, `v = xl_state(0, prec, 8)` creates a vector of one zero as content and with 8 as the maximum length.

Conceptually, a vector state variable is a double ended queue. It has two ends, the front which is the element at address 0 and the back which is the element at length – 1.

Methods available for vector are:

`val = v(idx);`	Returns the value of element at address idx.
`v(idx) = val;`	Assigns the element at address idx with val.
`f = v.front;`	Returns the value of the front end. An error is thrown if the vector is empty.
`v.push_front(val);`	Pushes val to the front and then increases the vector length by 1. An error is thrown if the vector is full.
`v.pop_front;`	Pops one element from the front and decreases the vector length by 1. An error is thrown if the vector is empty.
`b = v.back;`	Returns the value of the back end. An error is thrown if the vector is empty.

`v.push_back(val);`	Pushes val to the back and the increases the vector length by 1. An error is thrown if the vector is full.
`v.pop_back;`	Pops one element from the back and decreases the vector length by 1. An error is thrown if the vector is empty.
`v.push_front_pop_back(val);`	Pushes val to the front and pops one element out from the back. It's a shift operation. The length of the vector is unchanged. The vector cannot be empty to perform this operation.
`full = v.full;`	Returns `true` if the vector is full, otherwise, `false`.
`empty = v.empty;`	Returns `true` if the vector is empty, otherwise, `false`.
`len = v.length;`	Returns the number of elements in the vector.

A method of a vector that queries a state variable is called a *query method*. It has a return value. The following methods are query method: `v(idx)`, `v.front`, `v.back`, `v.full`, `v.empty`, `v.length`, `v.maxlen`. A method of a vector that changes a state variable is called an *update method*. An update method does not return any value. The following methods are update methods: `v(idx) = val`, `v.push_front(val)`, `v.pop_front`, `v.push_back(val)`, `v.pop_back`, and `v.push_front_pop_back(val)`. All query methods of a vector must be invoked before any update method is invocation during any simulation cycle. An error is thrown during model compilation if this rule is broken.

The MCode block can map a vector state variable into a vector of registers, a delay line, an addressable shift register, a single port ROM, or a single port RAM based on the usage of the state variable. The `xl_state` function can also be used to convert a MATLAB 1-D array into a zero-indexed constant array. If the MCode block cannot map a vector state variable into an FPGA, an error message is issued during model netlist time. The following are examples of using vector state variables.

Delay Line

The state variable in the following function is mapped into a delay line.

```
function q = delay(d, lat)
  persistent r, r = xl_state(zeros(1, lat), d, lat);
  q = r.back;
  r.push_front_pop_back(d);
```

Line of Registers

The state variable in the following function is mapped into a line of registers.

```
function s = sum4(d)
  persistent r, r = xl_state(zeros(1, 4), d);
  S = r(0) + r(1) + r(2) + r(3);
  r.push_front_pop_back(d);
```

Vector of Constants

The state variable in the following function is mapped into a vector of constants.

```
function s = myadd(a, b, c, d, nbits, binpt)
  p = {xlSigned, nbits, binpt, xlRound, xlSaturate};
  persistent coef, coef = xl_state([3, 7, 3.5, 6.7], p);
  s = a*coef(0) + b*coef(1) + c*coef(2) + c*coef(3);
```

Addressable Shift Register

The state variable in the following function is mapped into an addressable shift register.

```
function q = addrsr(d, addr, en, depth)
  persistent r, r = xl_state(zeros(1, depth), d);
  q = r(addr);
  if en
 r.push_front_pop_back(d);
  end
```

Single Port ROM

The state variable in the following function is mapped into a single port ROM.

```
function q = addrsr(contents, addr, arith, nbits, binpt)
  proto = {arith, nbits, binpt};
  persistent mem, mem = xl_state(contents, proto);
  q = mem(addr);
```

- **Single Port RAM:**

  The state variable in the following function is mapped to a single port RAM in fabric (Distributed RAM).

  ```
  function dout = ram(addr, we, din, depth, nbits, binpt)
    proto = {xlSigned, nbits, binpt};
    persistent mem, mem = xl_state(zeros(1, depth), proto);
    dout = mem(addr);
    if we
      mem(addr) = din;
    end
  ```

  The state variable in the following function is mapped to BlockRAM as a single port RAM.

  ```
  function dout = ram(addr, we, din, depth, nbits, binpt,ram_enable)
    proto = {xlSigned, nbits, binpt};
    persistent mem, mem = xl_state(zeros(1, depth), proto);
    persistent dout_temp, dout_temp = xl_state(0,proto);
    dout = dout_temp;
  ```

```
  dout_temp = mem(addr);
  if we
    mem(addr) = din;
  end
```

MATLAB Functions

- **disp() :**

  Displays the expression value. In order to see the printing on the MATLAB console, the option **Enable printing with disp** must be checked on the **Advanced** tab of the MCode block parameters dialog box. The argument can be a string, an `xfix` number, or an MCode state variable. If the argument is an `xfix` number, it will print the type, binary value, and double precision value. For example, if variable `x` is assigned with `xfix({xlSigned, 10, 7}, 2.75)`, the `disp(x)` will print the following line:

  ```
  type: Fix_10_7, binary: 010.1100000, double: 2.75
  ```

  If the argument is a vector state variable, `disp()` will print out the type, maximum length, current length, and the binary and double values of all the elements. For each simulation step, when **Enable printing with disp** is on and when a **disp()** function is invoked, a title line is printed for the corresponding block. The title line includes the block name, Simulink simulation time, and FPGA clock number.

  The following MCode function shows several examples of using the `disp()` function.

  ```
  function x = testdisp(a, b)
  persistent dly, dly = xl_state(zeros(1, 8), a);
  persistent rom, rom = xl_state([3, 2, 1, 0], a);
  disp('Hello World!');
  disp(['num2str(dly) is ', num2str(dly)]);
  disp('disp(dly) is ');
  disp(dly);
  disp('disp(rom) is ');
  disp(rom);
  a2 = dly.back;
  dly.push_front_pop_back(a);
  x = a + b;
  disp(['a = ', num2str(a), ', ', ...
  'b = ', num2str(b), ', ', ...
  'x = ', num2str(x)]);
  disp(num2str(true));
  disp('disp(10) is');
  disp(10);
  disp('disp(-10) is');
  disp(-10);
  disp('disp(a) is ');
  disp(a);
  disp('disp(a == b)');
  disp(a==b);
  ```

The following lines are the result for the first simulation step.

```
xlmcode_testdisp/MCode (Simulink time: 0.000000, FPGA clock: 0)
Hello World!
num2str(dly) is [0.000000, 0.000000, 0.000000, 0.000000, 0.000000,
0.000000,
0.000000, 0.000000]
disp(dly) is
type: Fix_11_7,
maxlen: 8,
length: 8,
0: binary 0000.0000000, double 0.000000,
1: binary 0000.0000000, double 0.000000,
2: binary 0000.0000000, double 0.000000,
3: binary 0000.0000000, double 0.000000,
4: binary 0000.0000000, double 0.000000,
5: binary 0000.0000000, double 0.000000,
6: binary 0000.0000000, double 0.000000,
7: binary 0000.0000000, double 0.000000,
disp(rom) is
type: Fix_11_7,
maxlen: 4,
length: 4,
0: binary 0011.0000000, double 3.0,
1: binary 0010.0000000, double 2.0,
2: binary 0001.0000000, double 1.0,
3: binary 0000.0000000, double 0.0,
a = 0.000000, b = 0.000000, x = 0.000000
1
disp(10) is
type: UFix_4_0, binary: 1010, double: 10.0
disp(-10) is
type: Fix_5_0, binary: 10110, double: -10.0
disp(a) is
type: Fix_11_7, binary: 0000.0000000, double: 0.000000
disp(a == b)
type: Bool, binary: 1, double: 1
```

- **error() :**

  Displays message and abort function. See MATLAB help on this function for more detailed information. Message formatting is not supported by the MCode block. For example:

```
if latency <=0
  error('latency must be a positive');
end
```

- **isnan() :**

  Returns true for Not-a-Number. `isnan(X)` returns true when `X` is Not-a-Number. `X` must be a scalar value of double or Xilinx fixed-point number. This function is not supported for vectors or matrices. For example:

```
if isnan(incr) & incr == 1
  cnt = cnt + 1;
end
```

- **NaN() :**

  The `NaN()` function generates an IEEE arithmetic representation for Not-a-Number. A NaN is obtained as a result of mathematically undefined operations like 0.0/0.0 and inf-inf. NaN(1,N) generates a 1-by-N vector of NaN values. Here are examples of using NaN.

```
if x < 0
  z = NaN;
else
  z = x + y;
end
```

- **num2Str() :**

  Converts a number to a string. `num2str(X)` converts the `X` into a string. `X` can be a scalar value of double, a Xilinx fixed-point number, or a vector state variable. The default number of digits is based on the magnitude of the elements of `X`. Here's an example of `num2str`:

```
if opcode <=0 | opcode >= 10
  error(['opcode is out of range: ', num2str(opcode)]);
end
```

- **ones():**

  The `ones()` function generates a specified number of one values. `ones(1,N)` generates a 1-by-N vector of ones. `ones(M,N)` where `M` must be 1. It's usually used with `xl_state()` function call. For example, the following line creates a 1-by-4 vector state variable initialized to [1, 1, 1, 1].

```
persitent m, m = xl_state(ones(1, 4), proto)
```

- **zeros() :**

  The `zeros()` function generates a specified number of zero values. `zeros(1,N)` generates a 1-by-N vector of zeros. `zero(M,N)` where `M` must be 1. It's usually used with `xl_state()` function call. For example, the following line creates a 1-by-4 vector state variable initialized to [0, 0, 0, 0].

```
persitent m, m = xl_state(zeros(1, 4), proto)
```

- **FOR Loop:**

FOR statement is fully unrolled. The following function sums n samples.

```
function q = sum(din, n)
  persistent regs, regs = xl_state(zeros(1, 4), din);
  q = reg(0);
  for i = 1:n-1
    q = q + reg(i);
  end
  regs.push_front_pop_back(din);
```

The following function does a bit reverse.

```
function q = bitreverse(d)
  q = xl_slice(d, 0, 0);
  for i = 1:xl_nbits(d)-1
    q = xl_concat(q, xl_slice(d, i, i));
  end
```

- **Variable Availability :**

  MATLAB code is sequential (for example, statements are executed in order). The MCode block requires that every possible execution path assigns a value to a variable before it is used (except as a left-hand side of an assignment). When this is the case, we say the variable is *available* for use. The MCode block will throw an error if its M-code function accesses unavailable variables.

  Consider the following M-code:

```
function [x, y, z] = test1(a, b)
  x = a;
  if a>b
    x = a + b; y = a;
  end
  switch a
    case 0
      z = a + b;
    case 1
      z = a - b;
  end
```

  Here a, b, and x are available, but y and z are not. Variable y is not available because the **if** statement has no **else**, and variable z is not available because the **switch** statement has no **otherwise** part.

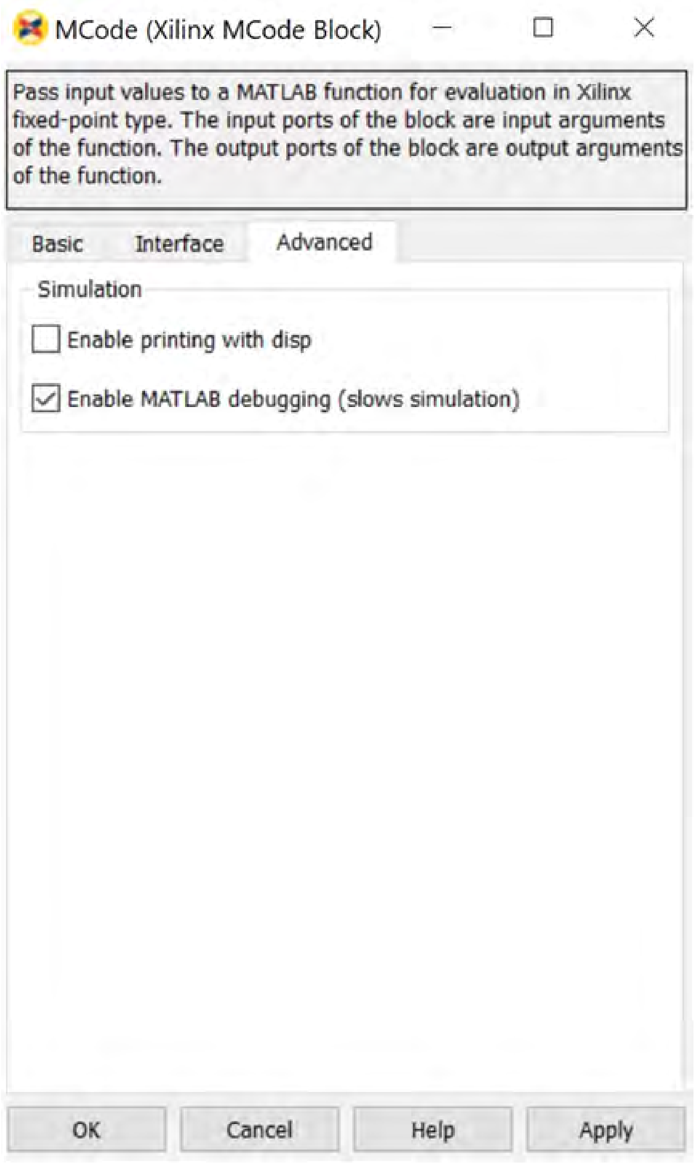
  DEBUG MCode

  There are two ways to debug your MCode. One is to insert disp() functions in your code and enable printing; the other is to use the MATLAB debugger. For usage of the disp() function, see the disp() section in this topic.

If you want to use the MATLAB debugger, you need to check the **Enable MATLAB debugging** option on the **Advanced** tab of the MCode block parameters dialog box. Then you can open your MATLAB function with the MATLAB editor, set break points, and debug your M-function. Just be aware that every time you modify your script, you need to execute a `clear functions` command in the MATLAB console.

To start debugging your M-function, you need to first check the **Enable MATLAB debugging** check box on the **Advanced** tab of the MCode block parameters dialog, then click the **OK** or **Apply** button.

Figure 346: **Enable MATLAB Debugging**


MCode (Xilinx MCode Block)

Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

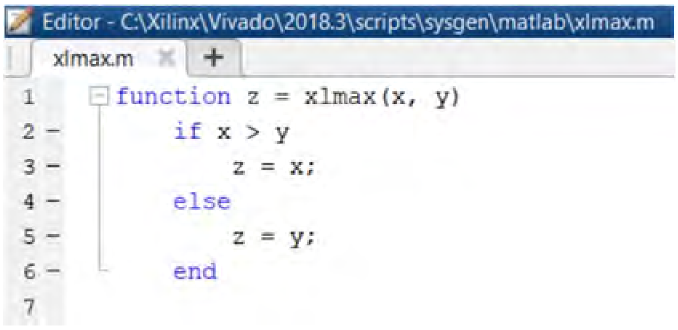
Basic | Interface | Advanced

Simulation

☐ Enable printing with disp

☑ Enable MATLAB debugging (slows simulation)

Now you can edit the M-file with the MATLAB editor and set break points as needed.

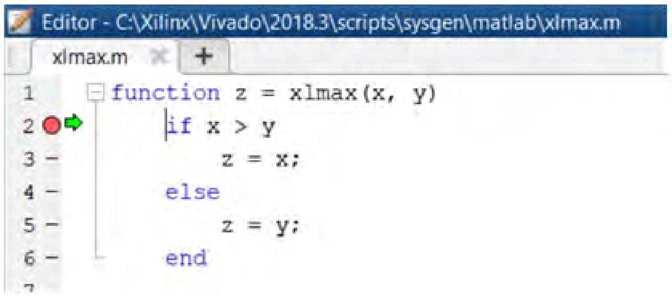


Editor - C:\Xilinx\Vivado\2018.3\scripts\sysgen\matlab\xlmax.m

```
function z = xlmax(x, y)
    if x > y
        z = x;
    else
        z = y;
    end
```

Figure 347: **Set Break Points**

During the Simulink simulation, the MATLAB debugger will stop at the break points you set when the break points are reached.

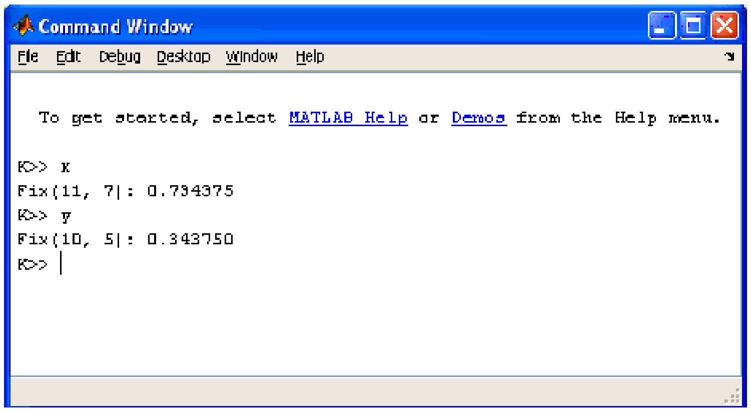


Editor - C:\Xilinx\Vivado\2018.3\scripts\sysgen\matlab\xlmax.m

```
function z = xlmax(x, y)
    if x > y
        z = x;
    else
        z = y;
    end
```

Figure 348: **Stopping at Break Point**

When debugging, you can also examine the values of the variables by typing the variable names in the MATLAB console.



```
  To get started, select MATLAB Help or Demos from the Help menu.

K>> x
Fix(11, 7|: 0.734375
K>> y
Fix(10, 5|: 0.343750
K>> 
```

Figure 349: **Examining Variable**

There is one special case to consider when the function for an MCode block is executed from the MATLAB debugger. A `switch/case` expression inside an MCode block must be type `xfix`, however, executing a `switch/case` expression from the MATLAB console requires that the expression be a `double` or `char`. To facilitate execution in the MATLAB console, a call to `double()` must be added. For example, consider the following:

```
switch i
  case 0
    x = 1
  case 1
    x = 2
  end
```

where `i` is type `xfix`. To run from the console this code must changed to

```
switch double(i)
  case 0
    x = 1
  case 1
    x = 2
end
```

The `double()` function call only has an effect when the M code is run from the console. The MCode block ignores the `double()` call.

- **Passing Parameters:**

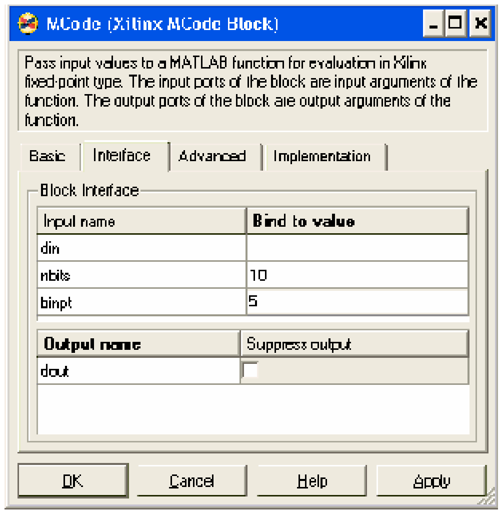
  It is possible to use the same M-function in different MCode blocks, passing different parameters to the M-function so that each block can behave differently. This is achieved by binding input arguments to some values. To bind the input arguments, select the **Interface** tab on the block GUI. After you bind those arguments to some values, these M-function arguments will not be shown as input ports of the MCode block.

  Consider for example, the following M-function:

```
function dout = xl_sconvert(din, nbits, binpt)
proto = {xlSigned, nbits, binpt};
dout = xfix(proto, din);
```

  The following figures shows how the bindings are set for the **din** input of two separate `xl_sconvert` blocks.

Figure 350: **din Bindings, Example 1**



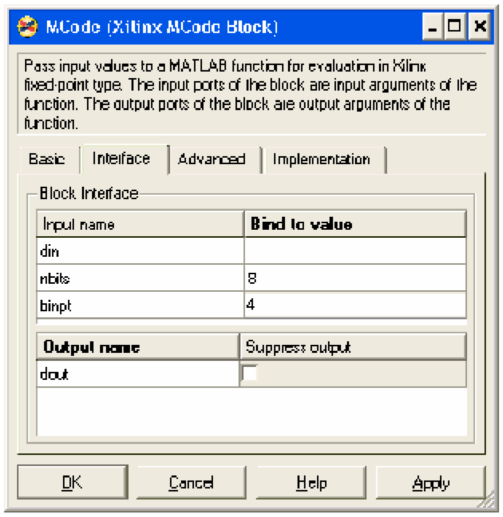
MCode (Xilinx MCode Block)

Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

Input name	Bind to value
din	
nbits	10
binpt	5

Output name	Suppress output
dout	☐

Figure 351: **din Bindings, Example 2**



MCode (Xilinx MCode Block)

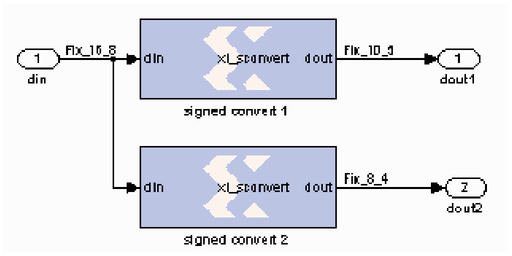
Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

Input name	Bind to value
din	
nbits	8
binpt	4

Output name	Suppress output
dout	☐

The following figure shows the block diagram after the model is compiled.

Figure 352: **Block Diagram**



The parameters can only be of type double or they can be logical numbers.

- **Optional Input Ports:**

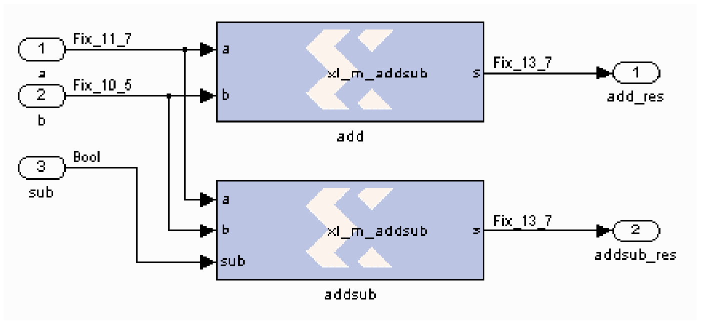
  The parameter passing mechanism allows the MCode block to have optional input ports. Consider for example, the following M-function:

```
function s = xl_m_addsub(a, b, sub)
  if sub
    s = a - b;
  else
    s = a + b;
  end
```

  If `sub` is set to be `false`, the MCode block that uses this M-function will have two input ports `a` and `b` and will perform full precision addition. If it is set to an empty cell array {}, the block will have three input ports `a`, `b`, and `sub` and will perform full precision addition or subtraction based on the value of input port `sub`.

  The following figure shows the block diagram of two blocks using the same `xl_m_addsub` function, one having two input ports and one having three input ports.

Figure 353: **Two Blocks Using Same xl_m_addsub Function**



- **Constructing a State Machine:**

There are two ways to build a state machine using an MCode block. One way is to specify a stateless transition function using a MATLAB function and pair an MCode block with one or more state register blocks. Usually the MCode block drives a register with the value representing the next state, and the register feeds back the current state into the MCode block. For this to work, the precision of the state output from the MCode block must be static, that is, independent of any inputs to the block. Occasionally you might find you need to use `xfix()` conversions to force static precision. The following code illustrates this:

```
function nextstate = fsm1(currentstate, din)
  % some other code
  nextstate = currentstate;
  switch currentstate
    case 0, if din==1, nextstate = 1; end
  end
  % a xfix call should be used at the end
  nextstate = xfix({xlUnsigned, 2, 0}, nextstate);
```

Another way is to use state variables. The above function can be re-written as follows:

```
function currentstate = fsm1(din)
  persistent state, state=xl_state(0,{xlUnsigned,2,0});
  currentstate = state;
  switch double(state)
    case 0, if din==1; state = 1; end
  end
```

- **Reset and Enable Signals for State Variables:**

  The MCode block can automatically infer register reset and enable signals for state variables when conditional assignments to the variables contain two or fewer branches.

  For example, the following M-code infers an enable signal for conditional assignment of persistent state variable `r1`:

```
function myFn = aFn(en, a)
  persistent r1, r1 = xl_state(0, {xlUnsigned, 2, 0});
  myFn = r1;
  if en
    r1 = r1 + a
  else
    r1 = r1
  end
```

  There are two branches in the conditional assignment to persistent state variable `r1`. A register is used to perform the conditional assignment. The input of the register is connected to `r1 + a`, the output of the register is `r1`. The register's enable signal is inferred; the enable signal is connected to `en`, when `en` is asserted. Persistent state variable `r1` is assigned to `r1 + a` when `en` evaluates to `false`, the enable signal on the register is de-asserted resulting in the assignment of `r1` to `r1`.

The following M-code will also infer an enable signal on the register used to perform the conditional assignment:

```
function myFn = aFn(en, a)
  persistent r1, r1 = xl_state(0, {xlUnsigned, 2, 0});
  myFn = r1;
  if en
    r1 = r1 + a
  end
```

An enable is inferred instead of a reset because the conditional assignment of persistent state variable `r1` is to a non-constant value, `r1 + a`.

If there were three branches in the conditional assignment of persistent state variable `r1`, the enable signal would not be inferred. The following M-code illustrates the case where there are three branches in the conditional assignment of persistent state variable `r1` and the enable signal is not inferred:

```
function myFn = aFn(en, en2, a, b)
  persistent r1, r1 = xl_state(0, {xlUnsigned, 2, 0});
  if en
    r1 = r1 + a
  elseif en2
    r1 = r1 + b
  else
    r1 = r1
  end
```

The reset signal can be inferred if a persistent state variable is conditionally assigned to a constant; the reset is synchronous. Consider the following M-code example which infers a reset signal for the assignment of persistent state variable `r1` to `init`, a constant, when `rst` evaluates to true and `r1 + 1` otherwise:

```
function myFn = aFn(rst)
  persistent r1, r1 = xl_state(0, {xlUnsigned, 4, 0});
  myFn = r1;
  init = 7;
  if (rst)
    r1 = init
  else
    r1 = r1 + 1
  end
```

The M-code example above which infers reset can also be written as:

```
function myFn = aFn(rst)
  persistent r1, r1 = xl_state(0, {xlUnsigned,4,0});
  init = 1;
  myFn = r1;
  r1 = r1 +1
  if (rst)
    r1 = init
  end
```

In both code examples above, the reset signal of the register containing persistent state variable `r1` is assigned to `rst`. When `rst` evaluates to `true`, the register's reset input is asserted and the persistent state variable is assigned to constant `init`. When `rst` evaluates to `false`, the register's reset input is de-asserted and persistent state variable `r1` is assigned to `r1 + 1`. Again, if the conditional assignment of a persistent state variable contains three or more branches, a reset signal is not inferred on the persistent state variable's register.

It is possible to infer reset and enable signals on the register of a single persistent state variable. The following M-code example illustrates simultaneous inference of reset and enable signals for the persistent state variable `r1`:

```
function myFn = aFn(rst,en)
  persistent r1, r1 = xl_state(0, {xlUnsigned, 4, 0});
  myFn = r1;
  init = 0;
  if rst
    r1 = init
  else
    if en
      r1 = r1 + 1
    end
  end
```

The reset input for the register of persistent state variable `r1` is connected to `rst`; when `rst` evaluates to `true`, the register's reset input is asserted and `r1` is assigned to `init`. The enable input of the register is connected to `en`; when `en` evaluates to `true`, the register's enable input is asserted and `r1` is assigned to `r1 + 1`. It is important to note that an inferred reset signal takes precedence over an inferred enable signal regardless of the order of the conditional assignment statements. Consider the second code example above; if both `rst` and `en` evaluate to `true`, persistent state variable `r1` would be assigned to `init`.

Inference of reset and enable signals also works for conditional assignment of persistent state variables using switch statements, provided the switch statements contain two or less branches.

The MCode block performs dead code elimination and constant propagation compiler optimizations when generating code for the FPGA. This can result in the inference of reset and/or enable signals in conditional assignment of persistent state variables, when one of the branches is never executed. For this to occur, the conditional must contain two branches that are executed after dead code is eliminated, and constant propagation is performed.

- **Inferring Registers :**

  Registers are inferred in hardware by using persistent variables, however, the right coding style must be used. Consider the two code segments in the following function:

  ```
  function [out1, out2] = persistent_test02(in1, in2)
  persistent ff1, ff1 = xl_state(0, {xlUnsigned, 2, 0});
  persistent ff2, ff2 = xl_state(0, {xlUnsigned, 2, 0});
  %code segment 1
  out1 = ff1; %these two statements infer a register for ff1
  ff1  = in1;
  %code segment 2
  ff2  = in2; %these two statements do NOT infer a register for ff2
  out2 = ff2;
  end
  ```

  In code segment 1, the value of persistent variable ff1 is assigned to out1. Since ff1 is persistent , it is assumed that its current value was assigned in the previous cycle. In the next statement, the value of in1 is assigned to ff1 so it can be saved for the next cycle. This infers a register for ff1.

  In code segment 2, the value of in2 is first assigned to persistent variable ff2, then assigned to out2. These two statements can be completed in one cycle, so a register is not inferred. If you need to insert delay into combinational logic, refer to the next topic.

- **Pipelining Combinational Logic :**

  The generated FPGA bitstream for an MCode block might contain many levels of combinational logic and hence a large critical path delay. To allow a downstream logic synthesis tool to automatically pipeline the combinational logic, you can add delay blocks before the MCode block inputs or after the MCode block outputs. These delay blocks should have the parameter **Implement using behavioral HDL** set, which instructs the code generator to implement delay with synthesizable HDL. You can then instruct the downstream logic synthesis tool to implement register re-timing or register balancing. As an alternative approach, you can use the vector state variables to model delays.

- **Shift Operations with Multiplication and Division:**

The MCode block can detect when a number is multiplied or divided by constants that are powers of two. If detected, the MCode block will perform a shift operation. For example, multiplying by 4 is equivalent to left shifting 2 bits and dividing by 8 is equivalent to right shifting 3 bits. A shift is implemented by adjusting the binary point, expanding the `xfix` container as needed. For example, a `Fix_8_4` number multiplied by 4 will result in a `Fix_8_2` number, and a `Fix_8_4` number multiplied by 64 will result in a `Fix_10_0` number.

- **Using the xl_state Function with Rounding Mode:**

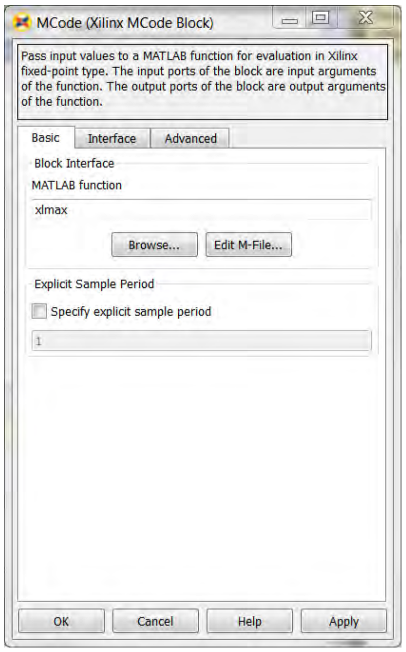
  The `xl_state` function call creates an `xfix` container for the state variable. The container's precision is specified by the second argument passed to the `xl_state` function call. If precision uses `xlRound` for its rounding mode, hardware resources is added to accomplish the rounding. If rounding the initial value is all that is required, an `xfix` call to round a constant does not require additional hardware resources. The rounded value can then be passed to the `xl_state` function. For example:

```
init = xfix({xlSigned,8,5,xlRound,xlWrap}, 3.14159);
persistent s, s = xl_state(init, {xlSigned, 8, 5});
```

Block Parameters

The block parameters dialog box can be invoked by double-clicking the block icon in a Simulink® model.

Figure 354: **Block Parameters**


MCode (Xilinx MCode Block)

Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.

Block Interface

MATLAB function: xlmax

Explicit Sample Period

Specify explicit sample period: [ ]

1

As described earlier in this topic, the **MATLAB function** parameter on an MCode block tells the name of the block's function, and the **Interface** tab specifies a list of constant inputs and their values.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Questa

The HDL Black Box block provides a way to incorporate existing HDL files into a model. When the model is simulated, co-simulation can be used to allow black boxes to participate. The Questa HDL co-simulation block configures and controls co-simulation for one or several black boxes.

During a simulation, each Questa block spawns one copy of Questa, and therefore uses one Questa license. If licenses are scarce, several black boxes can share the same block.

In detail, the Questa block does the following:

- Constructs the additional VHDL and Verilog needed to allow black box HDL to be simulated inside Questa.
- Spawns a Questa session when a Simulink simulation starts.
- Mediates the communication between Simulink and Questa.
- Reports if errors are detected when black box HDL is compiled.
- Terminates Questa, if appropriate, when the simulation is complete.

***Note*:** The Questa block only supports symbolic radix in the Questa tool. In symbolic radix, Questa displays the actual values of an enumerated type. and also converts an object's value to an appropriate representation for other radix forms. Please refer to the Questa documentation for more information on symbolic radix.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

- **Basic tab:** Parameters specific to the Basic tab are as follows.
  - **Run co-simulation in directory:**

Questa is started in the directory named by this field. The directory is created if necessary. All black box files are copied into this directory, as are the auxiliary files Model Composer produces for co-simulation. Existing files are overwritten silently. The directory can be specified as an absolute or relative path. Relative paths are interpreted with respect to the directory in which the Simulink .mdl file resides.

- **Open waveform viewer:**

  When this checkbox is selected, the Questa waveform window opens automatically, displaying a standard set of signals. The signals include all inputs and outputs of all black boxes and all clock and clock enable signals supplied by Model Composer. The signal display can be customized with an auxiliary tcl script. To specify the script, select Add Custom Scripts and enter the script name (e.g., myscript.do) in the **Script to Run After vsim** field.

- **Leave Questa open at end of simulation:**

  When this checkbox is selected, the Questa session is left open after the Simulink simulation has finished.

- **Skip compilation (use previous results):**

  When this checkbox is selected, the Questa compilation phase is skipped in its entirety for all black boxes that are using the Questa block for HDL co-simulation. To select this option is to assert that: (1) underneath the directory in which Questa will run, there exists a Questa work directory, and (2) that the work directory contains up-to-date Questa compilation results for all black box HDL. Selecting this option can greatly reduce the time required to start-up the simulation, however, if it is selected when inappropriate, the simulation can fail to run or run but produce false results.

- **Advanced tab:** Parameters specific to the Advanced tab are as follows.

  - **Include Verilog unisim library:**

    Selecting this checkbox ensures that Questa includes the Verilog UniSim library during simulation. Note: the Verilog unisim library must be mapped to UNISIMS_VER in Questa. In addition, selecting this checkbox ensures the "glbl.v" module is compiled and invoked during simulation.

  - **Add custom scripts:**

    The term “script” refers to a Tcl macro file (DO file) executed by Questa. Selecting this checkbox activates the fields **Script to Run Before Starting Compilation**, **Script to Run in Place of "vsim"**, and **Script to Run after "vsim"**. The DO file scripts named in these fields are not run unless this checkbox is selected.

  - **Script to run before starting compilation:**

Enter the name of a Tcl macro file (DO file) that is to be executed by Questa before compiling black box HDL files.

Note: For information on how to write a Questa macro file (DO file) refer to the **Tcl and macros (DO files)** section in the *Vitis Model Composer User Guide* (UG1483).

- **Script to run in place of "vsim":**

  Questa uses the Tcl (tool command language) as the scripting language for controlling and extending the tool. Enter the name of a Questa Tcl macro file (DO file) that is to be executed by the Questa `do` command at the point when Model Composer would ordinarily instruct Questa to begin a simulation. To start the simulation after the macro file starts executing, you must place a `vsim` command inside the macro file.

  Normally, if this parameter is left blank, or Add custom scripts is not selected, then Model Composer instructs Questa to execute the default command `vsim $toplevel -title {Model Composer Co-Simulation (from block $blockname}` Here `$toplevel` is the name of the top level entity for simulation (e.g., work.my_model_mti_block) and `$blockname` is the name of the Questa block in the Simulink model associated with the current co-simulation. To avoid problems, certain characters in the block name (e.g., newlines) are sanitized.

  If this parameter is not blank and **Add custom scripts** is selected, then Model Composer instead instructs Questa to execute `do $* $toplevel $blockname. Here $toplevel and $blockname` are as above and `$*` represents the literal text entered in the field. If, for example the literal text is '`foo.do`', then Questa executes `foo.do`. This macro file can then reference `$toplevel` and `$blockname` as $1 and $2, respectively. Thus, the command `vsim $1` inside of the macro file `foo.do` runs vsim on topLevel.

- **Script to run after "vsim":**

  Enter the name of a Tcl macro file (DO file) that is to be executed by Questa after all the HDL for black boxes has been successfully compiled, and after the Questa simulation has completed successfully. If the **Open Waveform Viewer** checkbox has been selected, Model Composer issues all commands it ordinarily uses to open and customize the waveform viewer before running this script. This allows you to customize the waveform viewer as desired (either by adding signals to the default viewer or by creating a fully custom viewer). The black box tutorial includes an example that customizes the waveform viewer.

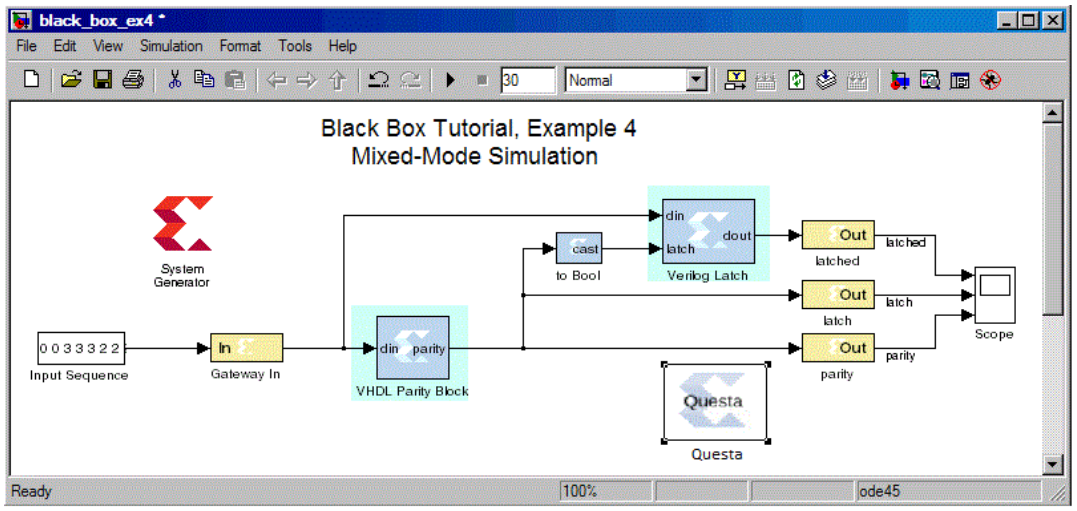
  It is often convenient to use relative paths in a custom script. Relative paths are interpreted with respect to the directory that contains the model's MDL file. A relative path in the Run co-simulation in directory field is also interpreted with respect to the directory that contains the model's MDL file. Thus, for example, if Run co-Simulation in directory specifies `./questa` as the directory in which Questa should run, the relative path ../foo.do in a script definition field refers to a file named foo.do in the directory that contains the .mdl.

Fine Points

The time scale in Questa matches that in Simulink. For example, one second of Simulink simulation time corresponds to one second of Questa simulation time. This makes it easy to compare times at which events occur in the two settings. The typically large Simulink time scale is also useful because it allows Model Composer to schedule events without running into problems related to the timing characteristics of the HDL model. You need not worry too much about the details of Model Composer event scheduling in co-simulation models.

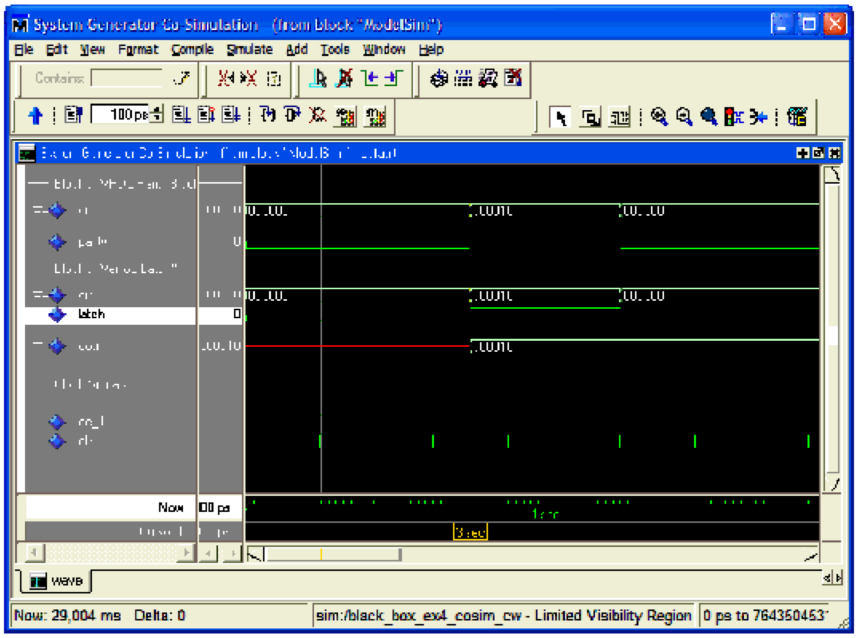
The following example is offered to illustrate the broader points.

Figure 355: **Example Model**



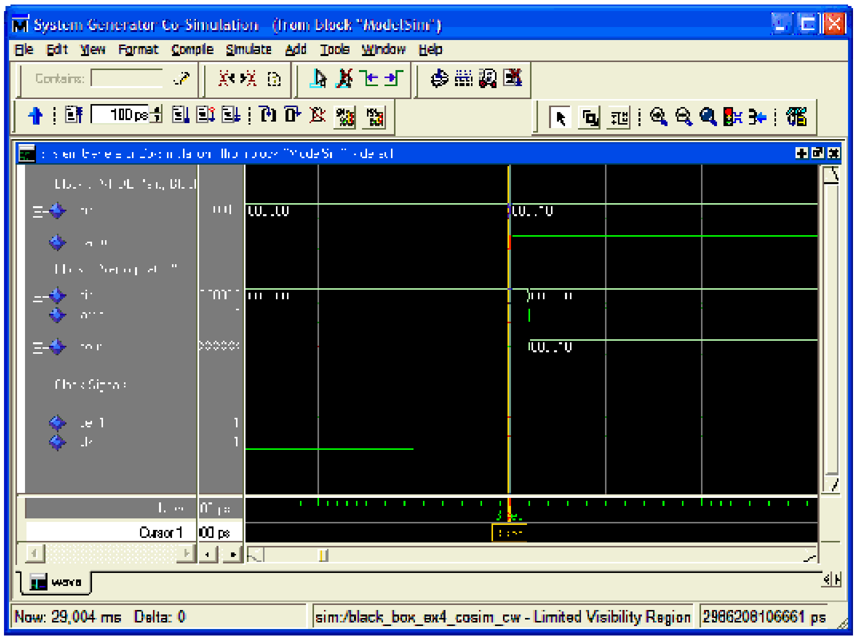
When the above model is run, the following waveforms are displayed by Questa:

Figure 356: **Example Time Scale**


System Generator Co-Simulation (from block "ModelSim")
File Edit View Format Compile Simulate Add Tools Window Help
latch
Now
wave
Now: 29,004 ms Delta: 0
sim:/black_box_ex4_cosim_cw - Limited Visibility Region
0 ps to 764350453

At the time scale presented here (the above shows a time interval of six seconds), the rising clock edge at three seconds and the corresponding transmission of data through each of the two black boxes appear simultaneous, much as they do in the Simulink simulation. Looking at the model, however, it is clear that the output of the first black box feeds the second black box. Both of the black boxes in this model have combinational feed-throughs, for example, changes on inputs translate into immediate changes on outputs. Zooming in toward the three second event reveals how Model Composer has resolved the dependencies. Note the displayed time interval has shrunk to ~20 ms.

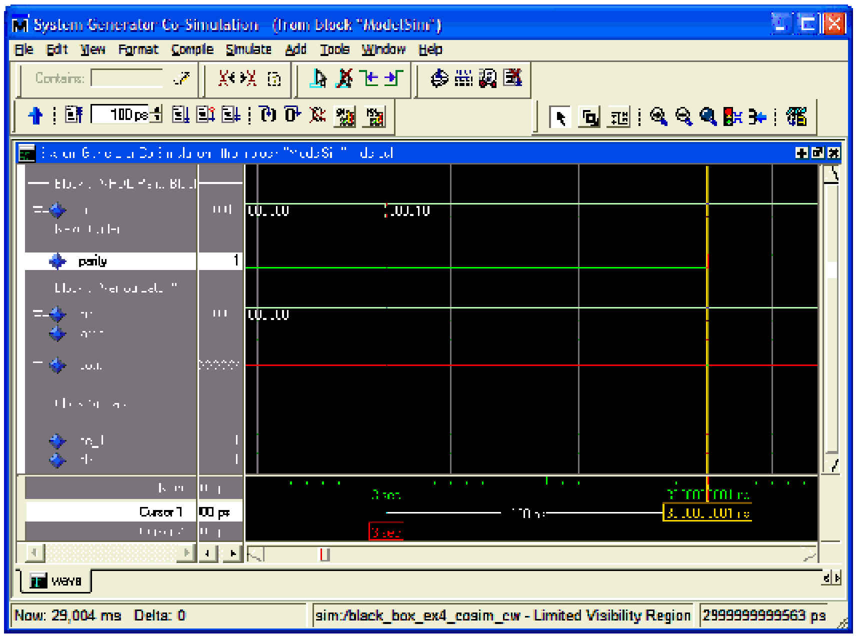
Figure 357: **Resolved Dependencies**


System Generator Co-Simulation (from block "ModelSim")
File Edit View Format Compile Simulate Add Tools Window Help
Contains:
100 ps
Cursor 1
wave

Now: 29,004 ms Delta: 0 sim:/black_box_ex4_cosim_cw - Limited Visibility Region 2986208106661 ps

The above figure reveals that Model Composer has shifted the rising clock edge so it occurs before the input value is collected from Simulink and presented to the first of the black boxes. It then allows the value to propagate through the first black box and presents the result to the second at a slightly later time. Zooming in still further shows that the HDL model for the first black box includes a propagation delay which Model Composer has effectively abstracted away through the use of large time scales. The actual delay through the first black box (exactly 1 ns) can be seen in the figure below.

Figure 358: **Delay Through the First Black Box**

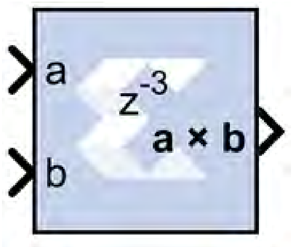

System Generator Co-Simulation (from block "ModelSim")
File Edit View Format Compile Simulate Add Tools Window Help
parity
wave

Now: 29,004 ms Delta: 0 | sim:/black_box_ex4_cosim_cw - Limited Visibility Region | 2999999999563 ps

In propagating data through black box components, Model Composer allocates 1/ 1000 of the system clock period down to 1us, then shrinks the allocation to 1/100 of the system clock period down to 5 ns, and below that threshold resorts to delta-delay stepping, for example, issuing "run 0 ns" commands to Questa. If the HDL includes timing information (e.g,. transport delays) and the Simulink System Period is set too low, then the simulation results are incorrect. The above model begins to fail when the Simulink system period setting is reduced below 5e-7, which is the point at which Model Composer resorts to delta-delay stepping of the black boxes for data propagation.

Mult

The Xilinx Mult block implements a multiplier. It computes the product of the data on its two input ports, producing the result on its output port.



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:** Parameters specific to the Basic tab are as follows.
    - **Precision:**

      This parameter allows you to specify the output precision for fixed-point arithmetic. Floating point output always has **Full** precision.

        - **Full:** The block uses sufficient precision to represent the result without error.
        - **User Defined:** If you do not need full precision, this option allows you to specify a reduced number of total bits and/or fractional bits.
    - **Fixed-point output type:**
        - **Arithematic Type:**
            - **Signed (2's comp):** The output is a Signed (2's complement) number.
            - **Unsigned:** The output is an Unsigned number.
            - **Number of bits:** Specifies the bit location of the binary point of the output number, where bit zero is the least significant bit.
            - **Binary point:** Position of the binary point in the fixed-point output.
    - **Quantization:**

      Refer to the Overflow and Quantization section in the Common Options in Block Parameter Dialog Boxes topic.
    - **Overflow:**

      Refer to the Overflow and Quantization section in the Common Options in Block Parameter Dialog Boxes topic.
    - **Optional Port:** Provide enable port
    - **Latency:** This defines the number of sample periods by which the block's output is delayed.

      Note: Only when latency of the Mult block is set to 4 in Model Composer, are all three pipeline stages used in the generated Multiplier IP.
- **Implementation tab:** Parameters specific to the Implementation tab are as follows.
    - **Use behavioral HDL (otherwise use core):**

The block is implemented using behavioral HDL. This gives the downstream logic synthesis tool maximum freedom to optimize for performance or area.

Note: For Floating-point operations, the block always uses the Floating-point Operator core.

- **Core Parameters:**
  - **Optimize for Speed|Area:** Directs the block to be optimized for either Speed or Area.
  - **Use embedded multipliers:** This field specifies that if possible, use the XtremeDSP slice (DSP48 type embedded multiplier) in the target device.
  - **Test for optimum pipelining:** Checks if the Latency provided is at least equal to the optimum pipeline length. Latency values that pass this test imply that the core produced is optimized for speed.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

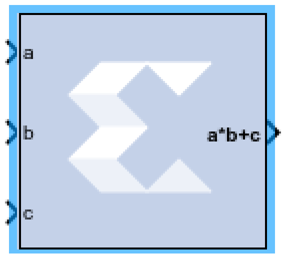
LogiCORE™ Documentation

LogiCORE IP Multiplier v12.0

LogiCORE IP Floating-Point Operator v7.1

MultAdd

The Xilinx MultAdd block performs both fixed-point and floating-point multiply and addition with the a and b inputs used for the multiplication and the c input for addition or subtraction.



If the MultAdd inputs are floating point, then inputs a,b, and c must be of the same data type. If the inputs are fixed point, then the port c binary point must be aligned to the sum of the port a and port b binary points.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:** Parameters specific to the Basic tab are as follows.
  - **Operation:**
    - **Addition:** Specifies that an addition will be performed after multiplication.
    - **Subtraction:** Specifies that a subtraction will be performed after multiplication.
    - **Addition or subtraction:** Adds a **subtract** port to the block, which controls whether the operation following multiplication is addition or subtraction (**subtract** High = subtraction, **subtract** Low = addition).
  - **Optional Ports:**
    - **Provide enable port:** Adds an active-High enable port to the block interface.
  - **Latency:**
    - **Latency:**

      This defines the number of sample periods by which the block's output is delayed. The latency values you can set depend on whether you are performing fixed point or floating point arithmetic:

      - For fixed point arithmetic, you can only specify a latency of **0** (for no latency) or **-1** (for maximum/optimal latency). If you have added an enable port to the block interface, you can only specify a latency of **-1** for fixed point arithmetic.
      - For floating point arithmetic, you can only specify a latency of **0** (for no latency) or a positive integer. If you have added an enable port to the block interface, you can only specify a positive integer for floating point arithmetic.

      See the LogiCORE IP Multiply Adder v3.0 Product Guide for details on latency in the block.
- **Output tab:**

  Parameters specific to the Output tab are as follows.

  - **Precision:** This parameter allows you to specify the output precision for fixed-point arithmetic. Floating point arithmetic output will always be **Full** precision.
    - **Full:** The block uses sufficient precision to represent the result without error.
    - **User Defined:** If you do not need full precision, this option allows you to specify a reduced number of total bits and/or fractional bits.
  - **Fixed-point Output Type:** Arithmetic type
    - **Signed (2’s comp):** The output is a Signed (2’s complement) number.
    - **Unsigned:** The output is an Unsigned number.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

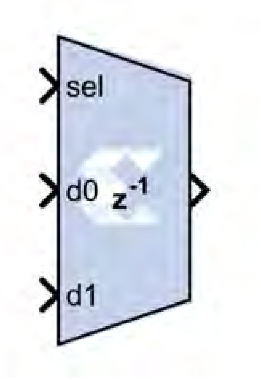
LogiCORE Documentation

LogiCORE IP Multiply Adder v3.0

LogiCORE IP Floating-Point Operator v7.1

Mux

The Xilinx Mux block implements a multiplexer. The block has one select input (type unsigned) and a user-configurable number of data bus inputs, ranging from 2 to 32.



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**
  - **Number of inputs:** specify a number between 2 and 32.
  - **Optional Ports:** Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.
- **Output tab:**
  - **Precision:** This parameter allows you to specify the output precision for fixed-point arithmetic. Floating point arithmetic output will always be **Full** precision.
    - **Full:** The block uses sufficient precision to represent the result without error.
    - **User Defined:** If you do not need full precision, this option allows you to specify a reduced number of total bits and/or fractional bits.

- **Fixed-point output type:**
  - **Arithmetic type:**
    - **Signed (2's comp):** The output is a Signed (2's complement) number.
    - **Unsigned:** The output is an Unsigned number.
    - **Number of bits:** Specifies the bit location of the binary point of the output number where bit zero is the least significant bit.
    - **Binary point:** Position of the binary point. in the fixed-point output.
- **Quantization:**

  Refer to the section Overflow and Quantization in the topic Common Options in Block Parameter Dialog Boxes.
- **Overflow:**

  Refer to the section Overflow and Quantization in the topic Common Options in Block Parameter Dialog Boxes.

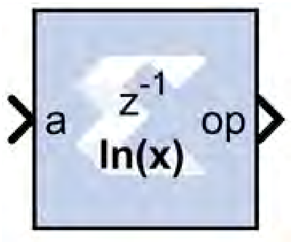
Parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP Floating-Point Operator v7.1

Natural Logarithm

The Xilinx Natural Logarithm block produces the natural logarithm of the input.



Block Parameters Dialog Box

- **Basic tab:** Parameters specific to the Basic tab are as follows.
  - **Flow Control Options:**

        - **Blocking:** In this mode, the block waits for data on the input, as indicated by TREADY, which allows back-pressure.
        - **NonBlocking:** In this mode, the block operates every cycle in which the input is valid, no back-pressure.

- **Optional Ports tab:**

  Parameters specific to the Basic tab are as follows.

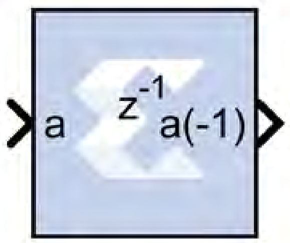
    - **Input Channel Ports:**
        - **Has TLAST:** Adds a tlast input port to the block.
        - **Has TUSER:** Adds a tuser input port to the block.
    - **Control Options:**
        - **Provide enable port:** Adds an enable port to the block interface.
        - **Has Result TREADY:** Adds a TREADY port to the output channel.
    - **Exception Signals::**
        - **INVALID_OP:** Adds an output port that serves as an invalid operation flag.
        - **DIVIDE_BY_ZERO:** Adds an output port that serves as a divide-by-zero flag.

LogiCORE Documentation

LogiCORE IP Floating-Point Operator v7.1

Negate

The Xilinx Negate block computes the arithmetic negation of its input.



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:** Parameters specific to the Basic tab are as follows.
  - **Precision:** This parameter allows you to specify the output precision for fixed-point arithmetic. Floating point output always has **Full** precision.
    - **Full:** The block uses sufficient precision to represent the result without error.
    - **User Defined:** If you do not need full precision, this option allows you to specify a reduced number of total bits and/or fractional bits.
    - **Fixed-Point Output Type:**

      Arithmetic Type
    - **Signed (2's comp):** The output is a Signed (2's complement) number.
    - **Unsigned:** The output is an Unsigned number.
  - **Fixed-point Precision:**
    - **Number of bits:** Specifies the bit location of the binary point of the output number, where bit zero is the least significant bit.
    - **Binary point:** Position of the binary point in the fixed-point output.
  - **Quantization:** Refer to the section Overflow and Quantization in the topic Common Options in Block Parameter Dialog Boxes.
  - **Overflow:** Refer to the section Overflow and Quantization in the topic Common Options in Block Parameter Dialog Boxes.
  - **Optional Port:** Provide enable port
  - **Latency:** This defines the number of sample periods by which the block's output is delayed.

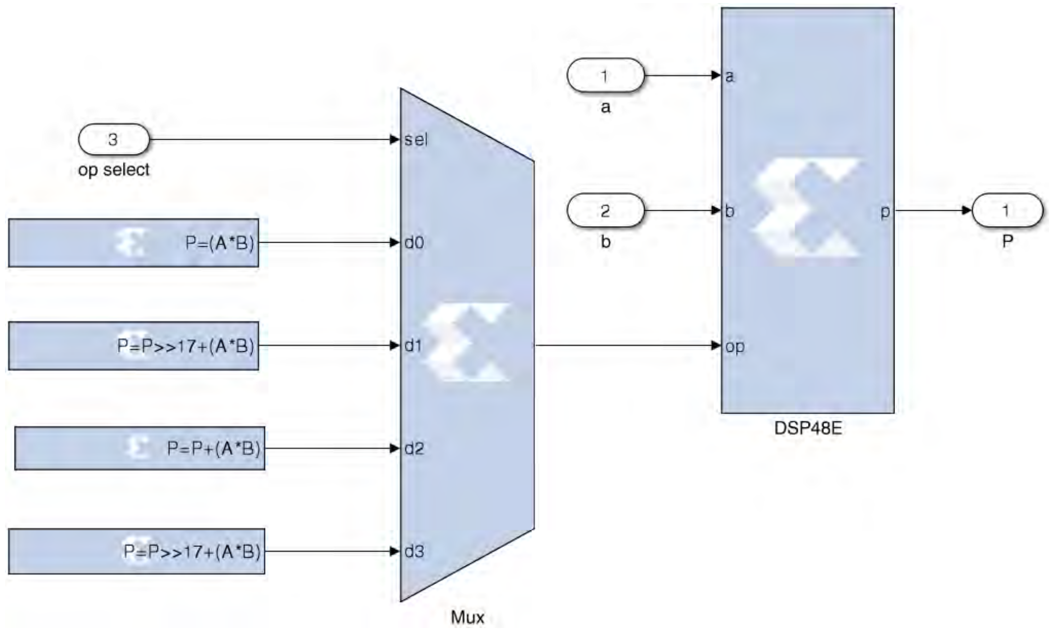
Parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Opmode

The Xilinx Opmode block generates a constant that is a DSP48E, DSP48E1, or DSP48E2 instruction. It is a 15-bit instruction for DSP48E, a 20-bit instruction for DSP48E1, and a 22-bit instruction for DSP48E2. The instruction consists of the opmode, carry-in, carry-in select, alumode, and (for DSP48E1 and DSP48E2) the inmode bits.

The Opmode block is useful for generating DSP48E, DSP48E1, or DSP48E2 control sequences. The figure below shows an example. The example implements a 35x35-bit multiplier using a sequence of four instructions in a DSP48E block. The Opmode blocks supply the desired instructions to a multiplexer that selects each instruction in the desired sequence.

Figure 359: **DSP48E Block**



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Opmode tab:**

  Parameters specific to the Opmode tab are as follows.

  - **Instruction:**

    - **Device:** Specifies whether to generate an instruction for the DSP48E, DSP48E1, or DSP48E2 device.

  - **DSP Instruction:**

    - **Operation:** Displays the instruction that is generated by the block. This instruction is also displayed on the block in the Simulink model.

    - **Operation select:** Selects the instruction.

    - **Preadder output:** Allows you to select the equation for the DSP48E1 Preadder.

      - **DSP Primitive Configuration:**

        - **Multiplier Output:** Allows you to select the Multiplier Output of DSP58 to be normal or negated.
        - **Preadder/Mult Function:** Allows you to select the function performed by the DSP48E2 Preadder/Multiplier.
        - **PREADDINSEL:** Displays the setting of the PREADDINSEL static control bits that are part of the instruction generated by the Opmode block. In the DSP48E2 slice, the PREADDDINSEL setting (A or B) selects the input to be added with the D input in the pre-adder.
        - **AMULTSEL:** Displays the setting of the AMULTSEL static control bits that are part of the instruction generated by the Opmode block. In the DSP48E2 slice, the AMULTSEL setting (A or AD) selects the input to the 27-bit A input of the multiplier.
        - **BMULTSEL:** Displays the setting of the BMULTSEL static control bits that are part of the instruction generated by the Opmode block. In the DSP48E2 slice, the BMULTSEL setting (B or AD) selects the input to the 18-bit B input of the multiplier.
    - **A register configuration:** Allows you to select the A register configuration for the DSP48E2. Select either A1 or A2.
    - **B register configuration:** Allows you to select the B register configuration for the DSP48E1 or DSP48E2. Select either B1 or B2.
- **Custom Instruction:**

  Note: The Custom Instruction field is activated when you select "Custom" in the **Operation select** field.

  - **Instruction:** Allows you to select the instruction for the DSP48E, DSP48E1, or DSP48E2.
  - **Z mux:** Specifies the 'Z' source to the add/sub/logic unit to be one of {'0', 'C', 'PCIN', 'P','C', 'PCIN>>17',' P>>17'}.
  - **XY muxes:** Specifies the 'XY' source to the DSP48's adder to be one of {'0','P', 'A:B', 'A*B', 'C', 'P+C', 'A:B+C' }. 'A:B' implies that A is concatenated with B to produce a value to be used as an input to the add/sub/logic unit.
  - **W mux:** Specifies the 'W' source to the DSP48E2's adder to be one of {'0','P', 'RND', 'C' }.
  - **Carry input:** Specifies the 'carry' source to the DSP48's add/sub/logic unit to be one of {'0', '1', 'CIN', 'Round PCIN towards infinity', 'Round PCIN towards zero', 'Round P towards infinity', 'Round P towards zero', 'Larger add/sub/acc (parallel operation)', 'Larger add/sub/acc (sequential operation)', 'Round A*B'}.

  For a description of any of the Custom Instruction options, see the following manuals:

- DSP48E: Virtex-5 FPGA XtremeDSP Design Considerations (UG193)
- DSP48E1: 7 Series DSP48E1 Slice User Guide (UG479)
- DSP48E2: UltraScale Architecture DSP Slice User Guide (UG579)

Xilinx LogiCORE

The Opmode block does not use a Xilinx LogiCORE.

DSP48E Control Instruction Format

DSP48E Instruction

Operation select	Notes
C + A*B	
PCIN + A*B	
P + A*B	
A* B	
C + A:B	
C - A:B	
C	
Custom	Use equation described in the Custom Instruction Field.

DSP48E Custom Instruction

Instruction Field Name	Location	Mnemonic	Notes
XY muxes	op[3:0]	0	0
		P	DSP48 output register
		A:B	Concat inputs A and B (A is MSB)
		A*B	Multiplication of inputs A and B
		C	DSP48 input C
		P+C	DSP48 input C plus P
		A:B+C	Concat inputs A and B plus C register
Z mux	op[6:4	0	0
		PCIN	DSP48 cascaded input from PCOUT
		P	DSP48 output register
		C	DSP48 C input
		PCIN>>17	Cascaded input downshifted by 17
		P>>17	DSP48 output register downshifted by 17

Instruction Field Name	Location	Mnemonic	Notes
Alumode	op[10:7]	X+Z	Add
		Z-X	Subtract
Carry input	op[14:12]	0 or 1	Set carry in to 0 or 1
		CIN	Select cin as source. This adds a CIN port to the Opmode block whose value is inserted into the mnemonic at bit location 11.
		Round PCIN toward infinity	
		Round PCIN toward zero	
		Round P toward infinity	
		Round P toward zero	
		Larger add/sub/acc (parallel operation)	
		Larger add/sub/acc (sequential operation)	
		Round A*B	

DSP48E1 Control Instruction Format

DSP48E1 Instruction

Operation select	Notes
C + A*B	
PCIN + A*B	
P + A*B	
A* B	
C + A:B	
C - A:B	
C	
Custom	Use equation described in the Custom Instruction Field.

Preadder output	Notes
Zero	
A2	
A1	
D + A2	
D + A1	
D	
-A2	
-A1	
D - A2	
D - A1	

B register configuration	Notes
B1	
B2	

DSP48E1 Custom Instruction

Instruction Field Name	Location	Mnemonic	Notes
Instruction		X + Z	
		X +NOT(Z)	
		NOT(X+Z)	
		Z - X	
		X XOR Z	
		X XNOR Z	
		X AND Z	
		X OR Z	
		X AND NOT(Z)	
		X OR NOT (Z)	
		X NAND Z	
Z mux	op[6:4]	0	0
		PCIN	DSP48 cascaded input from PCOUT
		P	DSP48 output register
		C	DSP48 C input
		PCIN>>17	Cascaded input downshifted by 17
		P>>17	DSP48 output register downshifted by 17
Operand: (Alumode)	op[10:7]	X+Z	Add
		Z-X	Subtract
XY muxes	op[3:0]	0	0
		P	DSP48 output register
		A:B	Concat inputs A and B (A is MSB)
		A*B	Multiplication of inputs A and B
		C	DSP48 input C
		P+C	DSP48 input C plus P
		A:B+C	Concat inputs A and B plus C register

Instruction Field Name	Location	Mnemonic	Notes
Carry input	op[14:12]	0 or 1	Set carry in to 0 or 1
		CIN	Select cin as source. This adds a CIN port to the Opmode block whose value is inserted into the mnemonic at bit location 11.
		Round PCIN toward infinity	
		Round PCIN toward zero	
		Round P toward infinity	
		Round P toward zero	
		Larger add/sub/acc (parallel operation)	
		Larger add/sub/acc (sequential operation)	
		Round A*B	

DSP48E2 Control Instruction Format

DSP48E2 Instruction

Operation select	Notes
C + A*B	
PCIN + A*B	
P + A*B	
A* B	
C + A:B	
C - A:B	
C	
Custom	Use equation described in the Custom Instruction Field.

Preadder/Mult Function	Notes
Zero	
A*B	
(D+A)*B	
(D-A)*B	
(D+A)**2	
(D-A)**2	
D**2	
A**2	
-(A**2)	
(D+A)*A	
(D-A)*A	
(D+B)*A	

Preadder/Mult Function	Notes
(D-B)*A	
D*A	
B*A	
-B*A	
(D+B)**2	
(D-B)**2	
B**2	
-(B**2)	
(D+B)*B	
(D-B)*B	

A register configuration	Notes
A1	
A2	

B register configuration	Notes
B1	
B2	

DSP48E2 Custom Instruction

Instruction Field Name	Location	Mnemonic	Notes
XY muxes	op[3:0]	0	0
		P	DSP48 output register
		A:B	Concat inputs A and B (A is MSB)
		A*B	Multiplication of inputs A and B
		C	DSP48 input C
		P+C	DSP48 input C plus P
		A:B+C	Concat inputs A and B plus C register
Z mux	op[6:4]	0	0
		PCIN	DSP48 cascaded input from PCOUT
		P	DSP48 output register
		C	DSP48 C input
		PCIN>>17	Cascaded input downshifted by 17
		P>>17	DSP48 output register downshifted by 17

Instruction Field Name	Location	Mnemonic	Notes
W mux	op[8:7]	0	
		P	DSP48 output register
		RND	Rounding Constant into W mux
		C	DSP48 input C
ALU mode (Instruction)	op[12:9]	X + W + Z	
		X +W + NOT(Z)	
		NOT(X + W + Z)	
		Z - (X+W)	
		X XOR Z	
		X XNOR Z	
		X AND Z	
		X OR Z	
		X AND NOT(Z)	
		X OR NOT (Z)	
		X NAND Z	
		X NOR Z	
		NOT (X) OR Z	
		NOT (X) AND Z	
Carry input	op[16:13]	0 or 1	Set carry in to 0 or 1
		CIN	Select CIN as source. This adds a CIN port to the Opmode block whose value is inserted into the mnemonic at bit location 11.
		Round PCIN toward infinity	
		Round PCIN toward zero	
		Round P toward infinity	
		Round P toward zero	
		Larger add/sub/acc (parallel operation)	
		Larger add/sub/acc (sequential operation)	
		Round A*B	

Instruction Field Name	Location	Mnemonic	Notes
Pre-Adder/Mult Function	op[21:17]	Zero	
		A * B	
		(D + A) * B	
		(D - A) * B	
		(D + A)**2	
		(D - A)**2	
		D**2	
		A**2	
		-(A**2)	
		(D + A) * A	
		(D - A) * A	
		(D + B) * A	
		(D - B) * A	
		D * A	
		B * A	
		-B * A	
		(D + B)**2	
		(D - B)**2	
		B**2	
		-(B**2)	
		(D + B) * B	
		(D - B) * B	

DSPCPLX Control Instruction Format

The OpMode block, when configured with a DSPCPLX device, outputs 18-bit wide data to drive the consolidated control port of the DSPCPLX block. The DSPCPLX block internally feeds the same setting to respective individual Real and Imaginary control input ports.

Table 57: **DSPCPLX Instruction**

Operation Select	Notes
A*B	
Custom	Use the equation described in the Custom Instruction Field.

Table 58: **DSPCPLX Custom Instruction**

Instruction Field Name	Location	Mnemonic	Notes
XY Muxes	op[3:0]	A*B	Multiplication of inputs A and B

Table 58: **DSPCPLX Custom Instruction** *(cont'd)*

Instruction Field Name	Location	Mnemonic	Notes
Z mux	op[6:4]	0	0
		PCIN	DSPCPLX cascaded input from PCOUT
		P	DSPCPLX output register
		C	DSPCPLX C input
W mux	op[8:7]	0	
		RND	Rounding constant into W mux
ALU mode (Instruction)	op[12:9]	X + W + Z	
Carry input	op[16:13]	0 or 1	Set carry in to 0 or 1
		CIN	Select CIN as source. This adds a CIN port to the Opmode block whose value is inserted into the mnemonic at bit location 11.
Conjugate A Input	op[17]	0 or 1	Select to complex conjugate the input A value
Conjugate B Input	op[18]	0 or 1	Select to complex conjugate the input B value

DSP58 Control Instruction Format

DSP58 Instruction

Operation select	Notes
C + A*B	
PCIN + A*B	
P + A*B	
A* B	
C + A:B	
C - A:B	
C	
Custom	Use equation described in the Custom Instruction Field.

Preadder/Mult Function	Notes
Zero	
±A*B	
±(D±A)*B	
±(D±A)**2	
±(D)**2	
±(A)**2	
±(D±A)*A	
±(D±B)*A	
±D*A	

Preadder/Mult Function	Notes
±(D±B)**2	
±(B)**2	
±(D±B)*B	

A register configuration	Notes
A1	
A2	

B register configuration	Notes
B1	
B2	

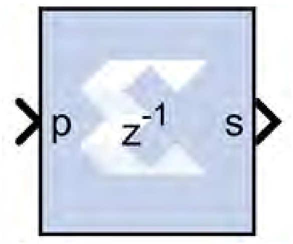
DSP58 Custom Instruction

Instruction Field Name	Location	Mnemonic	Notes
XY muxes	op[3:0]	0	0
		P	DSP58 output register
		A:B	Concat inputs A and B (A is MSB)
		A*B	Multiplication of inputs A and B
		C	DSP58 input C
		P+C	DSP58 input C plus P
		A:B+C	Concat inputs A and B plus C register
Z mux	op[6:4]	0	0
		PCIN	DSP58 cascaded input from PCOUT
		P	DSP58 output register
		C	DSP58 C input
		PCIN>>17	Cascaded input downshifted by 17
		P>>17	DSP58 output register downshifted by 17
W mux	op[8:7]	0	
		P	DSP58 output register
		RND	Rounding Constant into W mux
		C	DSP58 input C

Instruction Field Name	Location	Mnemonic	Notes
ALU mode (Instruction)	op[12:9]	X + W + Z	
		X +W + NOT(Z)	
		NOT(X + W + Z)	
		Z - (X+W)	
		X XOR Z	
		X XNOR Z	
		X AND Z	
		X OR Z	
		X AND NOT(Z)	
		X OR NOT (Z)	
		X NAND Z	
		X NOR Z	
		NOT (X) OR Z	
		NOT (X) AND Z	
Carry input	op[16:13]	0 or 1	Set carry in to 0 or 1
		CIN	Select CIN as source. This adds a CIN port to the Opmode block whose value is inserted into the mnemonic at bit location 11.
		Round PCIN toward infinity	
		Round PCIN toward zero	
		Round P toward infinity	
		Round P toward zero	
		Larger add/sub/acc (parallel operation)	
		Larger add/sub/acc (sequential operation)	
		Round A*B	
Pre-Adder/Mult Function	op[21:17]	Zero	
		±A*B	
		±(D±A)*B	
		±(D±A)**2	
		±(D)**2	
		±(A)**2	
		±(D±A)*A	
		±(D±B)*A	
		±D*A	
		±(D±B)**2	
		±(B)**2	
		±(D±B)*B	
Negate	op[24:22]	0 or 1	Select Negate to negate the Multiplier output

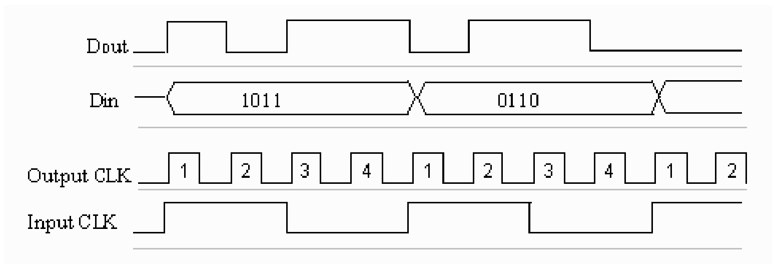
Parallel to Serial

The Parallel to Serial block takes an input word and splits it into N time-multiplexed output words where N is the ratio of number of input bits to output bits. The order of the output can be either least significant bit first or most significant bit first.



The following waveform illustrates the block's behavior:

Figure 360: **Block Behavior**



This example illustrates the case where the input width is 4, output word size is 1, and the block is configured to output the most significant word first.

Block Interface

The Parallel to Serial block has one input and one output port. The input port can be any size. The output port size is indicated on the block parameters dialog box.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

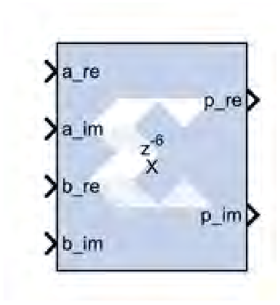
- **Basic tab:** Parameters specific to the Basic tab are as follows.
  - **Output order:** Most significant word first or least significant word first.
  - **Type:** Signed or unsigned.
  - **Number of bits:** Output width. Must divide Number of Input Bits evenly.
  - **Binary Point:** Binary point location.

The minimum latency of this block is 0.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Product

The Xilinx Product block implements a scalar or complex multiplier. It computes the product of the data on its two input channels, producing the result on its output channel. For complex multiplication the input and output have two components: real and imaginary.



The Product block is ideal for generating a simple scalar or complex multiplier. If your implementation will use more complicated features such as AXI4 ports or a user-specified precision, use the Xilinx Complex Multiplier 6.0 block (if you are configuring a complex multiplier) or Xilinx Mult block (if you are configuring a scalar multiplier) in your design instead of the Product block.

In the Vivado® design flow, the Product block is inferred as "LogiCORE™ IP Complex Multiplier" (if you have configured the Product block for complex multiplication) or “LogiCORE IP Multiplier” (if you have configured the Product block for scalar multiplication) for code generation. Refer to the LogiCORE IP Complex Multiplier v6.0 Product Guide or the LogiCORE IP Multiplier v12.0 Product Guide for details about these LogiCORE IP.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

Parameters specific to the block are as follows.

- **Complex Multiplication:** Specifies mode of operation: scalar multiplier (**Complex Multiplication** deselected) or complex multiplier (**Complex Multiplication** selected).
- **Optimize for:** Specifies whether your design will be optimized for **Performance** or for **Resources** when it is implemented in the Xilinx FPGA or SoC device.

Based on the settings for **Complex Multiplication** and **Optimize for**, and rate and type propagation (from the input data width), the latency value of the block will be derived automatically for a fully pipelined circuit. This latency value will be displayed on the block in the Simulink model.

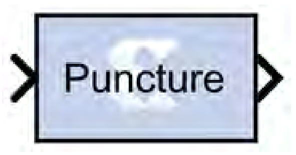
LogiCORE Documentation

LogiCORE IP Complex Multiplier v6.0

LogiCORE IP Multiplier v12.0

Puncture

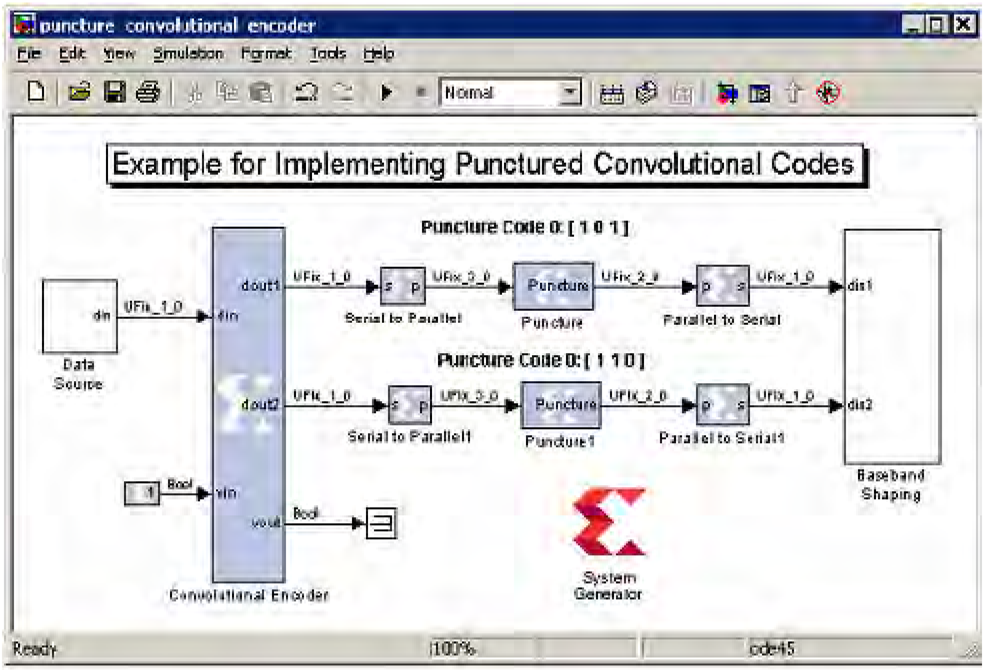
The Xilinx Puncture block removes a set of user-specified bits from the input words of its data stream.



Based on the puncture code parameter, a binary vector that specifies which bits to remove, it converts input data of type `UFixN_0` (where N is equal to the length of the puncture code) into output data of type `UFixK_0` (where K is equal to the number of ones in the puncture code). The output rate is identical to the input rate.

This block is commonly used in conjunction with a convolution encoder to implement punctured convolution codes as shown in the figure below.

Figure 361: **Implementing Punctured Convolution Codes**



The system shown implements a rate ½ convolution encoder whose outputs are punctured to produce four output bits for each three input bits. The top puncture block removes the center bit for code 0 ([1 0 1]) and bottom puncture block removes the least significant bit for code 1 ([1 1 0]), producing a 2-bit punctured output. These data streams are serialized into 1-bit in-phase and quadrature data streams for baseband shaping.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

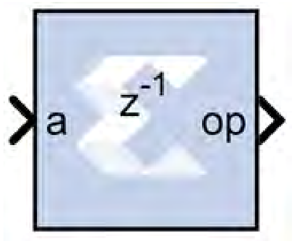
Parameters specific to the block are as follows:

- **Puncture Code:** The puncture pattern represented as a bit vector, where a zero in position i indicates bit i is to be removed.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Reciprocal

The Xilinx Reciprocal block performs the reciprocal on the input. Currently, only the floating-point data type is supported.



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

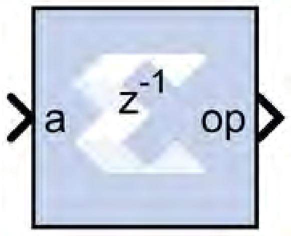
- **Basic tab:** Parameters specific to the Basic tab are as follows.
  - **Flow Control:**
    - **Blocking:** Selects "Blocking" mode. In this mode, the lack of data on one input channel does block the execution of an operation if data is received on another input channel.
    - **NonBlocking:** Selects "Non-Blocking" mode. In this mode, the lack of data on one input channel does not block the execution of an operation if data is received on another input channel.
- **Optional ports:**
  - **Input Channel Ports:**
    - **Has TLAST:** Adds a TLAST port to the Input channel.
    - **Has TUSER:** Adds a TUSER port to the Input channel.
    - **Provide enable port:** Adds an enable port to the block interface.
    - **Has Result TREADY:** Adds a TREADY port to the Result channel.
  - **Exception Signals:**
    - **UNDERFLOW:** Adds an output port that serves as an underflow flag.
    - **DIVIDE_BY_ZERO:** Adds an output port that serves as a divide-by-zero flag.

LogiCORE Documentation

LogiCORE IP Floating-Point Operator v7.1

Reciprocal SquareRoot

The Xilinx Reciprocal SquareRoot block performs the reciprocal squareroot on the input. Currently, only the floating-point data type is supported.



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows.

  - **Flow Control:**
    - **Blocking:** Selects Blocking mode. In this mode, the lack of data on one input channel does block the execution of an operation if data is received on another input channel.
    - **NonBlocking:** Selects Non-Blocking mode. In this mode, the lack of data on one input channel does not block the execution of an operation if data is received on another input channel.
- **Optional ports:**
  - **Input Channel Ports:**
    - **Has TLAST:** Adds a TLAST port to the Input channel.
    - **Has TUSER:** Adds a TUSER port to the Input channel.
    - **Provide enable port:** Adds an enable port to the block interface.
    - **Has Result TREADY:** Adds a TREADY port to the Result channel.
- **Exception Signals:**
  - **INVALID_OP:** Adds an output port that serves as an invalid operation flag.
  - **DIVIDE_BY_ZERO:** Adds an output port that serves as a divide-by-zero flag.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

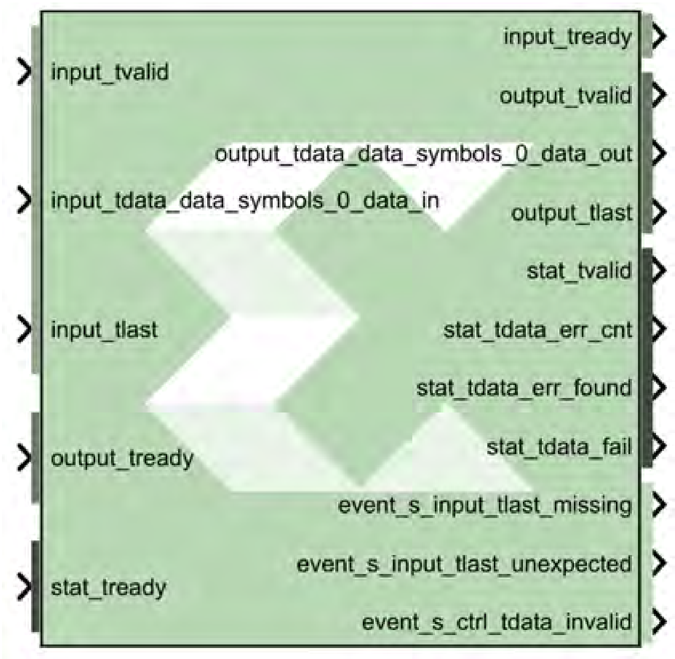
LogiCORE Documentation

LogiCORE IP Floating-Point Operator v7.1

Reed-Solomon Decoder 9.0

Note: This block goes into the FPGA fabric and is a Licensed Core. Please visit the Xilinx web site to purchase the appropriate core license.

The Reed-Solomon (RS) codes are block-based error correcting codes with a wide range of applications in digital communications and storage.


input_tvalid
input_tdata_data_symbols_0_data_in
input_tlast
output_tready
stat_tready
input_tready
output_tvalid
output_tdata_data_symbols_0_data_out
output_tlast
stat_tvalid
stat_tdata_err_cnt
stat_tdata_err_found
stat_tdata_fail
event_s_input_tlast_missing
event_s_input_tlast_unexpected
event_s_ctrl_tdata_invalid

They are used to correct errors in many systems such as digital storage devices, wireless/ mobile communications, and digital video broadcasting.

The Reed-Solomon decoder processes blocks generated by a Reed-Solomon encoder, attempting to correct errors, and recover information symbols. The number and type of errors that can be corrected depend on the characteristics of the code.

Reed-Solomon codes are Bose-Chaudhuri-Hocquenghem (BCH) codes, which in turn are linear block codes. An (*n*,*k*) linear block code is a *k*-dimensional sub-space of an *n*-dimensional vector space over a finite field. Elements of the field are called *symbols*. For a Reed-Solomon code, *n* ordinarily is 2^s-1, where *s* is the width in bits of each symbol. When the code is *shortened*, *n* is smaller. The decoder handles both full length and shortened codes. It is also able to handle *erasures*, that is, symbols that are known with high probability to contain errors.

When the decoder processes a block, there are three possibilities:

1. The information symbols are recovered. This is the case provided `2p+r <= n-k`, where `p` is the number of errors, and `r` is the number of erasures.
2. The decoder reports it is unable to recover the information symbols.
3. The decoder fails to recover the information symbols but does not report an error.

The probability of each possibility depends on the code and the nature of the communications channel. Simulink® provides excellent tools for modeling channels and estimating these probabilities.

Block Interface Channels and Pins

This Xilinx Reed-Solomon Decoder block is AXI4 compliant. The following describes the standard AXI channels and pins on the interface:

- **input Channel:**
  - **input_tvalid:** TVALID for the input channel.
  - **input_tdata_erase:** Added to the channel when you select **Erase** on the Optional Pins tab. It indicates the symbol currently presented on `data_in` should be treated as an erasure. The signal driving this pin must be `Bool`.
  - **input_tdata_data_in:** Presents blocks of `n` symbols to be decoded. This signal must have type `UFIX_s_0`, where `s` is the width in bits of each symbol.
  - **input_tlast:** Marks the last symbol of the input block. Only used to generate event outputs. Can be tied low or high if event outputs are not used.
  - **input_tready:** TREADY for the input channel.
  - **input_tuser_mark_in:** marker bits for tagging data on data_in. Added to the channel when you select **Marker Bits** from the Optional Pins tab.
- **output Channel:**
  - **output_tready:** TREADY for the output channel.
  - **output_tvalid:** TVALID for the output channel.
  - **output_tdata_data_out:** Produces the information and parity symbols resulting from decoding. The type of `data_out` is the same as that for `data_in`.
  - **output_tlast:** Goes high when the last symbol of the last block is on `tdata_data_out`. `output_tlast` produces a signal of type `UFIX_1_0`.
  - **output_tuser_mark_out:** mark_in tagging bits delayed by the latency of the LogiCORE™. Added to the channel when you select **Marker Bits** on the Optional Pins tab.
  - **output_tdata_info:** Added to the channel when you select **Info** on the Optional Pins tab. The signal marks the last information symbol of a block on `tdata_data_out`.

  - **output_tdata_data_del:** Added to the channel when you select **Original Delayed Data** on the Optional Pins tab. The signal marks the last information symbol of a block on `tdata_data_out`.

- **stat Channel:**

  - **stat_tready:** TREADY for the stat channel.

  - **stat_tvalid:** TVALID for the stat channel. You should tie this signal high if the downstream slave is always able to accept data or if the stat channel is not used.

  - **stat_tdata_err_cnt:** Presents a value at the time `data_out` presents the last symbol of the block. The value is the number of errors that were corrected. `err_cnt` must have type `UFIX_b_0` where `b` is the number of bits needed to represent n-k.

  - **stat_tdata_err_found:** presents a value at the time `output_tdata_data_out` presents the last symbol of the block. The value 1 if the decoder detected any errors or erasures during decoding. `err_found` must have type `UFIX_1_0`.

  - **stat_tdata_fail:** Presents a value at the time `output_tdata_data_out` presents the last symbol of the block. The value is 1 if the decoder was unable to recover the information symbols, and 0 otherwise. This signal must be of type `UFIX_1_0`.

  - **stat_tdata_erase_cnt:** Only available when erasure decoding is enabled. Presents a value at the time `dout` presents the last symbol of the block. The value is the number of erasures that were corrected This signal must be of type `UFIX_b_0` where `b` is the number of bits needed to represent n. Added to the channel when you select **Erase** from the Optional Pins tab.

  - **stat_tdata_bit_err_1_to_0:** Number of bits received as 1 but corrected to 0. Added to the channel when you select **Error Statistics** from the Optional Pins tab. The element width is the number of binary bits required to represent ((n-k) * Symbol_Width).

  - **stat_tdata_bit_err_0_to_1:** Number of bits received as 0 but corrected to 1. Added to the channel when you select **Error Statistics** from the Optional Pins tab. The element width is the number of binary bits required to represent ((n-k) * Symbol_Width).

  - **stat_tlast:** Added when Number of Channels parameter is greater than 1. Indicates that status information for the last channel is present on output_tdata.

- **event Channel:**

  - **event_s_input_tlast_missing:** This output flag indicates that the input_tlast was not asserted when expected. You should leave this pin unconnected if it is not required.

  - **event_s_input_tlast_unexpected:** This output flag indicates that the input_tlast was asserted when not expected. You should leave this pin unconnected if it is not required.

  - **event_s_ctrl_tdata_invalid:** This output flag indicates that values provided on `ctrl_tdata` were illegal. The block must be reset if this is asserted. You should leave this pin unconnected if it is not required.

- **ctrl Channel:** This channel is only present when variable block length, number of check symbols or puncture is selected as a block parameter.
  - **ctrl_tready:** TREADY for the ctrl channel.
  - **ctrl_tvalid_r_in:** TVALID for the ctrl channel.
  - **ctrl_tdata:** This input contains the block length, the number of check symbols and puncture select, if applicable.
- **Other Optional Pins:**
  - **aresetn:**

    Resets the decoder. This pin is added to the block when you specify **Synchronous Reset** on the Optional Pins tab. The signal driving `rst` must be `Bool`.

    `aresetn` must be asserted high for at least 1 sample period before the decoder can start decoding code symbols.
  - **aclken:** Carries the clock enable signal for the decoder. The signal driving `aclken` must be `Bool`. Added to the block when you select the optional pin **Clock Enable**.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Attributes 1 tab:** Parameters specific to the Attributes 1 tab are as follows.
  - **Code Block Specification:**
    - **Code specification:** Specifies the type of RS Decoder desired. The choices are as follows.
      - **Custom:** Allows you to set all the block parameters.
      - **DVB:** Implements DVB (Digital Video Broadcasting) standard (204, 188) shortened RS code.
      - **ATSC:** Implements ATSC (Advanced Television Systems Committee) standard (207, 187) shortened RS code.
      - **G.709:** Implements G.709 Optical Transport Network standard.
      - **CCSDS:** Implements CCSDS (Consultative Committee for Space Data Systems) standard (255, 223) full length RS code.
      - **IESS-308 (All):** Implements IESS-308 (INTELSAT Earth Station Standard) specification (all) shortened RS code.

- **IESS-308 (126):** Implements IESS-308 (INTELSAT Earth Station Standard) specification (126, 112) shortened RS code.
- **IESS-308 (194):** Implements IESS-308 specification (194, 178) shortened RS code.
- **IESS-308 (208):** Implements IESS-308 specification (208, 192) shortened RS code.
- **IESS-308 (219):** Implements IESS-308 specification (219, 201) shortened RS code.
- **IESS-308 (225):** Implements IESS-308 specification (225, 205) shortened RS code.
- **IEEE-802.16:** Implements IEEE-802.16 specification (255, 239) full length RS code.
- **Symbol width:** Tells the width in bits for symbols in the code. The encoder support widths from 3 to 12 (default 8).
- **Field polynomial:** Specifies the polynomial from which the symbol field is derived. It must be specified as a decimal number. This polynomial must be primitive. A value of zero indicates the default polynomial should be used. Default polynomials are listed in the table below.

Table 59: **Field Polynomials**

Symbol Width	Default Polynomials	Array Representation
3	$x^3 + x + 1$	11
4	$x^4 + x + 1$	19
5	x^5 + x2 + 1	37
6	$x^6 + x + 1$	67
7	$x^7 + x^3 + 1$	137
8	$x^8 + x^4 + x^3 + x^2 + 1$	285
9	$x^9 + x^4 + 1$	529
10	$x^{10} + x^3 + 1$	1033
11	$x^{11} + x^2 + 1$	2053
12	$x^{12} + x^6 + x^4 + x + 1$	4179

- **Scaling Factor (h):** (represented in the previous formula as h) specifies the scaling factor for the code. Ordinarily, h is 1, but can be as large as 2^S - 1 where s is the symbol width. The value must be chosen so that α^h is primitive. That is, h must be relatively prime to 2^S - 1.
- **Generator Start:** specifies the first root r of the generator polynomial. The generator polynomial g(x), is given by:

$$g(x) = \prod_{j=0}^{n-k-1} (x-\alpha^{h(\gamma+j)})$$

where α is a primitive element of the symbol field, and the scaling factor is described below.

- **Variable Block Length:** When checked, the block is given a `ctrl` input channel.
- **Symbols Per Block(n):** Tells the number of symbols in the blocks the encoder produces. Acceptable numbers range from 3 to 2^S -1, where s denotes the symbol width.
- **Data Symbols(k):** Tells the number of information symbols each block contains. Acceptable values range from max(n - 256, 1) to n - 2.

- **Variable Check Symbol Options:**
  - **Variable Number of Check Symbols (r):**
  - **Define Supported R_IN Values:**

    If only a subset of the possible values that could be sampled on R_IN is actually required, then it is possible to reduce the size of the core slightly. For example, for the Intelsat standard, the R_IN input is 5 bits wide but only requires r values of 14, 16, 18, and 20. The core size can be slightly reduced by defining only these four values to be supported. If any other value is sampled on R_IN, the core will not decode the data correctly.
  - **Number of Supported R_IN Values:** Specify the number of supported R_IN values.
  - **Supported R_IN Definition File:** This is a COE file that defines the R values to be supported. It has the following format: radix=10; legal_r_vector=14,16,18,20; The number of elements in the legal_r_vector must equal the specified **Number of Supported R_IN Values**.

- **Attributes 2 tab:**
  - **Implementation:**
    - **State Machine:**
      - **Self Recovering:** When checked, the block synchronously resets itself if it enters an illegal state.
    - **Memory Style:** Select between **Distributed**, **Block** and **Automatic** memory choices.
    - **Number Of Channels:** Specifies the number of separate time division multiplexed channels to be processed by the encoder. The encoder supports up to 128 channels.
    - **Output check symbols:** If selected, then the entire n symbols of each block are output on the output channel. If not selected, then only the k information symbols are output.
    - **Puncture Options:**

    - **Number of Puncture Patterns:** Specifies how many puncture patterns the LogiCORE needs to handle. It is set to 0 if puncturing is not required.
    - **Puncture Definition File:** Specifies the pathname of the puncture definition file that is used to define the puncture patterns.

      A relative pathname can be specified for a COE file in the current working directory. For example, the syntax is [cwd '/ieee802_16d_puncturing.coe'].

- **Optional pins tab:**
  - **Clock Enable:** Adds a **aclken** pin to the block. This signal carries the clock enable and must be of type Bool.
  - **Info:** Adds the **output_tdata_info** pin. Marks the last information symbol of a block on tdata_data_out.
  - **Synchronous Reset:** Adds a **aresetn** pin to the block. This signal resets the block and must be of type Bool. The signal must be asserted for at least 2 clock cycles, however, it does not have to be asserted before the decoder can start decoding.
  - **Original Delayed Data:** When checked, the block is given a `tdata_data_del` output. Indicates that a DAT_DEL field is in the output_tdata output.
  - **Erase:** When checked, the block is given an `input_tdata_erase` input pin.
  - **Error Statistics:** adds the following error statistics outputs:
    - **bit_err_0_to_1:** Number of bits received as 1 but corrected to 0.
    - **bit_err_1_to_0:** Number of bits received as 0 but corrected to 1.
  - **Marker Bits:** Adds the following pins to the block:
    - **input_tuser_mark_in:** Carries marker bits for tagging data on `input_tdata_data_in`.
    - **output_tuser_mark_out:** Mark_in tagging bits delayed by the latency of the LogiCORE.
  - **Number of Marker Bits:** Specifies the number of marker bits.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

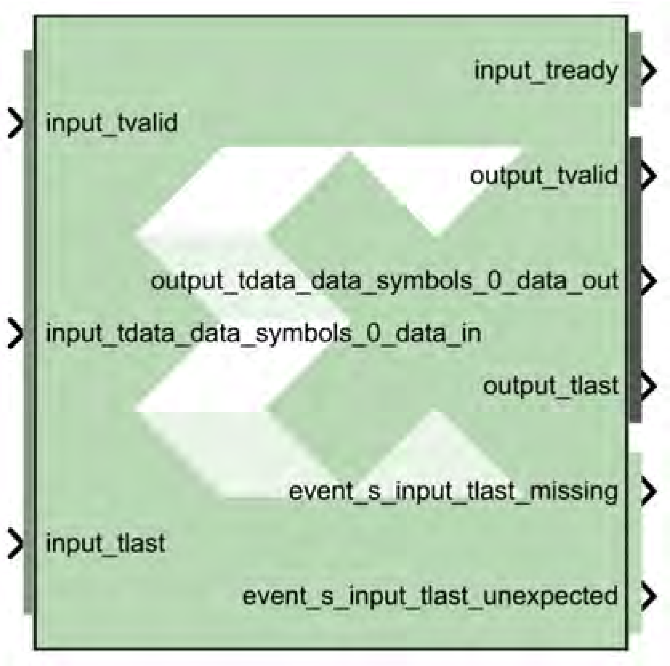
LogiCORE Documentation

LogiCORE IP Reed-Solomon Decoder v9.0

Reed-Solomon Encoder 9.0

Note: This block goes into the FPGA fabric and is a Licensed Core. Please visit the Xilinx web site to purchase the appropriate core license.

The Reed-Solomon (RS) codes are block-based error correcting codes with a wide range of applications in digital communications and storage. This block adheres to the AMBA® AXI4-StreamAXI4-Stream standard.

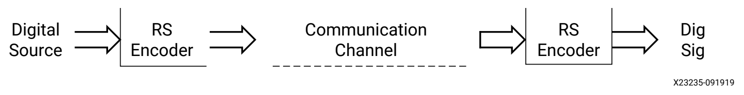

input_tready
input_tvalid
output_tvalid
output_tdata_data_symbols_0_data_out
input_tdata_data_symbols_0_data_in
output_tlast
event_s_input_tlast_missing
input_tlast
event_s_input_tlast_unexpected

They are used to correct errors in many systems such as digital storage devices, wireless or mobile communications, and digital video broadcasting.

The Reed-Solomon encoder augments data blocks with redundant symbols so that errors introduced during transmission can be corrected. Errors can occur for a number of reasons (noise or interference, scratches on a CD, etc.). The Reed-Solomon decoder attempts to correct errors and recover the original data. The number and type of errors that can be corrected depends on the characteristics of the code.

A typical system is shown below:

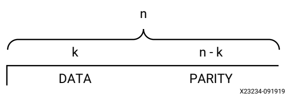
Figure 362: **Typical System**



Reed-Solomon codes are Bose-Chaudhuri-Hocquenghem (BCH) codes, which in turn are linear block codes. An (*n*, *k*) linear block code is a *k*-dimensional sub space of an *n*-dimensional vector space over a finite field. Elements of the field are called symbols. For a Reed-Solomon code, *n* ordinarily is 2^S -1, where s is the width in bits of each symbol. When the code is shortened, *n* is smaller. The encoder handles both full length and shortened codes.

The encoder is systematic. This means it constructs code blocks of length *n* from information blocks of length *k* by adjoining *n*-*k* parity symbols.

Figure 363: **Systematic Encoder**



A Reed-Solomon code is characterized by its field and generator polynomials. The field polynomial is used to construct the symbol field, and the generator polynomial is used to calculate parity symbols. The encoder allows both polynomials to be configured. The generator polynomial has the form:

$$g(x)=(x-\alpha^{j})(x-\alpha^{j+1}...(x-\alpha^{i+n-k-1})$$

where α is a primitive element of the finite field having n + 1 elements.

Block Interface Channels and Pins

The Xilinx Reed-Solomon Decoder 8.0 block is AXI4 compliant. The following describes the standard AXI channels and pins on the interface:

- **input Channel:**
  - **input_tvalid:** TVALID for the input channel.
  - **input_tdata_data_in:** Presents blocks of n symbols to be decoded. This signal must have type UFIX_s_0, where s is the width in bits of each symbol.
  - **input_tlast:** Marks the last symbol of the input block. Only used to generate event outputs. Can be tied low or high if event outputs are not used.
  - **input_tready:** TREADY for the input channel.
  - **input_tuser_marker:** Marker bits for tagging data on input_tdata_data_in. Added to the channel when you select **Marker Bits** from the Detailed Implementation tab.
- **output Channel:**
  - **output_tready:** TREADY for the output channel. Added to the channel when you select **Output TREADY** from the Optional Pins tab.

- **output_tvalid:** TVALID for the output channel.
- **output_tdata_data_out:** Produces the information and parity symbols resulting from decoding. The type of `data`_out is the same as that for `data_in`.
- **output_tlast:** Goes high when the last symbol of the last block is on `tdata_data_out`. `output_tlast` produces a signal of type `UFIX_1_0`.
- **output_tuser_maker:** This pin is available when user selects "Marker Bits" from the Detailed Implementation tab.

- **event Channel:**
    - **event_s_input_tlast_missing:** This output flag indicates that the input_tlast was not asserted when expected. You should leave this pin unconnected if it is not required.
    - **event_s_input_tlast_unexpected:** This output flag indicates that the input_tlast was asserted when not expected. You should leave this pin unconnected if it is not required.
    - **event_s_ctrl_tdata_invalid:** This output flag indicates that values provided on ctrl_tdata were illegal. This pin is available when "Variable Block Length" or "Variable Number of Check Symbols" are selected on the GUI.
- **ctrl Channel:** This channel is only present when variable block length or number of check symbols is selected as a block parameter.
    - **ctrl_tvalid:** TVALID for the ctrl channel.
    - **ctrl_tdata_n_in:** This signal is only present if “Variable Block Length” is selected in the GUI. This allows the block length to be changed every block. The ctrl_tdata_n_in signal must have type UFIX_s_0, where s is the width in bits of each symbol. Unless there is an R_IN field, the number of check symbols is fixed, so varying n automatically varies k.
    - **ctrl_tdata_r_in:** This field is only present if “Variable Number of Check Symbols” is selected in the GUI. It allows the number of check symbols to be changed every block. The new block's length, r_block, is set to ctrl_tdata_r_in sampled. The ctrl_tdata_r_in signal must have type UFIX_p_0, where p is the number of bits required to represent the parity bits (n-k) in the default code word, n being the "Symbols Per Block" and k being "Data Symbols". Selecting this input significantly increases the size of the core.

Other Optional Pins

- **aresetn:** Resets the encoder. This pin is added to the block when you specify **ARESETn** on the Detailed Implementation tab. The signal driving ARESETn must be Bool.

aresetn must be asserted low for at least 2 clock periods and at least 1 sample period before the decoder can start decoding code symbols.

- **aclken:** Carries the clock enable signal for the encoder. The signal driving aclken must be Bool. Added to the block when you select the optional pin **ACLKEN**.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Attributes:** Parameters specific to the Attributes tab are as follows.
    - **Code Block Specification:**
        - **Code specification:** specifies the encoder type desired. The choices are:
            - **Custom:** Allows you to set all the block parameters.
            - **DVB:** Implements DVB (Digital Video Broadcasting) standard (204, 188) shortened RS code.
            - **ATSC:** Implements ATSC (Advanced Television Systems Committee) standard (207, 187) shortened RS code
            - **G_709:** Implements G.709 Optical Transport Network standard.
            - **ETSI_BRAN:** Implements the ETSI Project standard for Broadband Radio Access Networks (BRAN).
            - **CCSDS:** Implements CCSDS (Consultative Committee for Space Data Systems) standard (255, 223) full length RS code.
            - **ITU_J_83_Annex_B:** Implements International Telecommunication Union(ITU)-J.83 Annex B specification (128, 122) extended RS code.
            - **IESS-308 (126):** Implements IESS-308 (INTELSAT Earth Station Standard) specification (126, 112) shortened RS code.
            - **IESS-308 (194):** Implements IESS-308 specification (194, 178) shortened RS code.
            - **IESS-308 (208):** Implements IESS-308 specification (208, 192) shortened RS code.
            - **IESS-308 (219):** Implements IESS-308 specification (219, 201) shortened RS code.
            - **IESS-308 (225):** Implements IESS-308 specification (225, 205) shortened RS code.
        - **Variable Number of Check Symbols (r):** False, true. When checked, the ctrl_tdata_r_in and ctrl_tdata_n_in pins become available on the block.
        - **Variable Block Length:** False, true. When checked, the ctrl_tdata_n_in pin becomes available on the block.
        - **Symbol width:** Tells the width in bits for symbols in the code. The encoder support widths from 3 to 12 and the default value is 8.

- **Field polynomial:** specifies the polynomial from which the symbol field is derived. It must be specified as a decimal number. This polynomial must be primitive. A value of zero indicates the default polynomial should be used. Default polynomials are listed in the table below.

  Table 60: **Field Polynomials**

Symbol Width	Default Polynomials	Array Representation
3	$x^3 + x + 1$	11
4	$x^4 + x + 1$	19
5	$x^5 + x2 + 1$	37
6	$x^6 + x + 1$	67
7	$x^7 + x^3 + 1$	137
8	$x^8 + x^4 + x^3 + x^2 + 1$	285
9	$x^9 + x^4 + 1$	529
10	$x^{10} + x^3 + 1$	1033
11	$x^{11} + x^2 + 1$	2053
12	$x^{12} + x^6 + x^4 + x + 1$	4179

- **Scaling Factor (h):** (represented in the previous formula as h) specifies the scaling factor for the code. Ordinarily, h is 1, but can be as large as 2^S - 1 where s is the symbol width. The value must be chosen so that α^h is primitive. That is, h must be relatively prime to 2^S - 1.

- **Generator Start:** Specifies the first root r of the generator polynomial. The generator polynomial g(x), is given by:

  $$g(x) = \prod_{j=0}^{n-k-1} (x - \alpha^{h(r+j)})$$

  where α is a primitive element of the symbol field, and the scaling factor is described below.

- **Symbols Per Block(n):** Tells the number of symbols in the blocks the encoder produces. Acceptable numbers range from 3 to 2^S -1, where s denotes the symbol width.

- **Data Symbols(k):** Tells the number of information symbols each block contains. Acceptable values range from max(n - 256, 1) to n - 2.

- **Detailed Implementation tab:**
  - **Implementation:**
    - **Check Symbol Generator Optimization:** This option is available when "Variable Number of Check Symbols" option is selected on the GUI.
      - **Fixed Architecture:** The check symbol generator is implemented using a highly efficient fixed architecture.

        - **Area:** The check symbol generator implementation is optimized for area and speed efficiency. The range of input, `ctrl_tdata_n_in`, is reduced.
        - **Flexibility:** The check symbol generator implementation is optimized to maximize the range of input of ctrl_tdata_n_in.
    - **Memory Style:** Select between **Distributed**, **Block** and **Automatic** memory choices. This option is available only for CCSDS codes.
    - **Number Of Channels:** Specifies the number of separate time division multiplexed channels to be processed by the encoder. The encoder supports up to 128 channels.
- **Optional Pins:**
    - **ACLKEN:** Adds a **aclken** pin to the block. This signal carries the clock enable and must be of type Bool.
    - **Output TREADY:** When selected, the output channels will have a TREADY and hence support the AXI4handshake protocol with inherent back-pressure.
    - **ARESETn:** Adds a **aresetn** pin to the block. This signal resets the block and must be of type Bool. aresetn must be asserted low for at least 2 clock periods and at least 1 sample period before the decoder can start decoding code symbols.
    - **Info bit:** Adds the **output_tdata_info** pin. Marks the last information symbol of a block on `tdata_data_out`.
    - **Marker Bits:** Adds the following pins to the block:
        - **input_tuser_user:** Carries marker bits for tagging data on `input_tdata_ data_in`.
        - **output_tuser_user:** `mark_in` tagging bits delayed by the latency of the LogiCORE.
    - **Number of Marker Bits:** Specifies the number of marker bits.

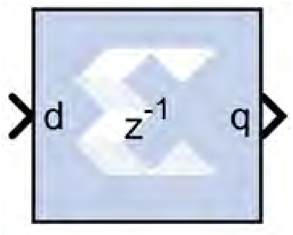
Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP Reed-Solomon Encoder v9.0

Register

The Xilinx Register block models a D flip-flop-based register, having latency of one sample period.



Block Interface

The block has one input port for the data and an optional input reset port. The initial output value is specified by you in the block parameters dialog box (below). Data presented at the input will appear at the output after one sample period. Upon reset, the register assumes the initial value specified in the parameters dialog box.

The Register block differs from the Xilinx Delay block by providing an optional reset port and a user specifiable initial value.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows.

  - **Initial value:** specifies the initial value in the register.
  - **Optional Ports:**

    Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Xilinx LogiCORE

The Register block is implemented as a synthesizable VHDL module. It does not use a Xilinx LogiCORE™.

Reinterpret

The Xilinx Reinterpret block forces its output to a new type without any regard for retaining the numerical value represented by the input.

The binary representation is passed through unchanged, so in hardware this block consumes no resources. The number of bits in the output will always be the same as the number of bits in the input.

The block allows for unsigned data to be reinterpreted as signed data, or, conversely, for signed data to be reinterpreted as unsigned. It also allows for the reinterpretation of the data's scaling, through the repositioning of the binary point within the data. The Xilinx Scale block provides an analogous capability.

An example of this block's use is as follows: if the input type is 6 bits wide and signed, with 2 fractional bits and the output type is forced to be unsigned with 0 fractional bits, then an input of -2.0 (1110.00 in binary, two's complement) would be translated into an output of 56 (111000 in binary).

This block can be particularly useful in applications that combine it with the Xilinx Slice block or the Xilinx Concat block. To illustrate the block's use, consider the following scenario:

Given two signals, one carrying signed data and the other carrying two unsigned bits (a UFix_2_0), we want to design a system that concatenates the two bits from the second signal onto the tail (least significant bits) of the signed signal.

We can do so using two Reinterpret blocks and one Concat block. The first Reinterpret block is used to force the signed input signal to be treated as an unsigned value with its binary point at zero. The result is then fed through the Concat block along with the other signal's UFix_2_0. The Concat operation is then followed by a second Reinterpret that forces the output of the Concat block back into a signed interpretation with the binary point appropriately repositioned.

Though three blocks are required in this construction, the hardware implementation is realized as simply a bus concatenation, which has no cost in hardware.

Block Parameters

Parameters specific to the block are as follows.

- **Force Arithmetic Type:** When checked, the Output Arithmetic Type parameter can be set and the output type is forced to the arithmetic type chosen according to the setting of the Output Arithmetic Type parameter. When unchecked, the arithmetic type of the output is unchanged from the arithmetic type of the input.
- **Output Arithmetic Type:** The arithmetic type (unsigned or signed, 2's complement, Floating-point) to which the output is to be forced.

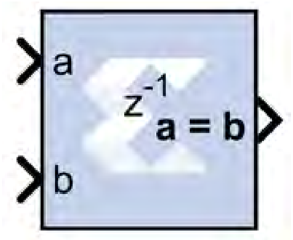
- **Force Binary Point:** When checked, the Output Binary Point parameter can be set and the binary point position of the output is forced to the position supplied in the Output Binary Point parameter. When unchecked, the arithmetic type of the output is unchanged from the arithmetic type of the input.
- **Output Binary Point:** The position to which the output's binary point is to be forced. The supplied value must be an integer between zero and the number of bits in the input (inclusive).

LogiCORE Documentation

LogiCORE IP Floating-Point Operator v7.1

Relational

The Xilinx Relational block implements a comparator.



The supported comparisons are the following:

- equal-to (a = b)
- not-equal-to (a != b)
- less-than (a < b)
- greater-than (a > b)
- less-than-or-equal-to (a <= b)
- greater-than-or-equal-to (a >= b)
- The output of the block is a `Bool`.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

The only parameter specific to the Relational block is:

- **Comparison**: specifies the comparison operation computed by the block.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP Floating-Point Operator v7.1

Requantize

The Xilinx Requantize block requantizes and scales its input signals.

The Xilinx Requantize block requantizes each input sample to a number of a desired fixed point precision output. For example, a fixed point signed (two's complement) or unsigned number can be requantized to an output with lesser or greater number of bits and realign its binary point precision.

This block also scales its input by a power of two. The power can be either positive or negative. The scale operation has the effect of moving the binary point without changing the bits in the container.

The Requantize block is used to requantize and scale its input signals. If you are only performing one of these operations, but not both, you can use a different block in the HDL blockset to perform that operation.

- To requantize your input without scaling, use the Convert block in the HDL blockset.
- To scale your input without requantizing, use the Scale block in the HDL blockset.

Quantization

Quantization errors occur when the number of fractional bits is insufficient to represent the fractional portion of a value. This block uses symmetric round during quantization for any insufficient input data.

Round (unbiased: +/- inf) is also known as "Symmetric Round (towards +/- inf)" or "Symmetric Round (away from zero)". This is similar to the MATLAB `round()` function. This method rounds the value to the nearest desired bit away from zero. When there is a value at the midpoint between two possible rounded values, the one with the larger magnitude is selected. For example, to round 01.0110 to a Fix_4_2, this yields 01.10, since 01.0110 is halfway between 01.01 and 01.10, and 01.10 is further from zero.

Overflow

Overflow errors occur when a value lies outside the representable range. In case of data overflow this block saturates the data to the largest positive/smallest negative value.

Block Parameters

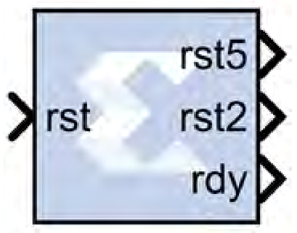
The block parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

Parameters specific to the block are as follows.

- **Scale factor s (scale output by 2^s):** The scale factor can be a positive or negative integer. The output of the block is `i`*2^`k`, where `i` is the input value and `k` is the scale factor. The effect of scaling is to move the binary point, which in hardware has no cost (a shift, on the other hand, might add logic).
- **Fixed-point Precision:**
  - **Number of bits:** Specifies the total number of bits, including the binary point bit width.
  - **Binary point:** Specifies the bit location of the binary point. Bit zero is the Least Significant Bit.

Reset Generator

The Reset Generator block captures the user's reset signal that is running at the system sample rate, and produces one or more downsampled reset signal(s) running at the rates specified on the block.

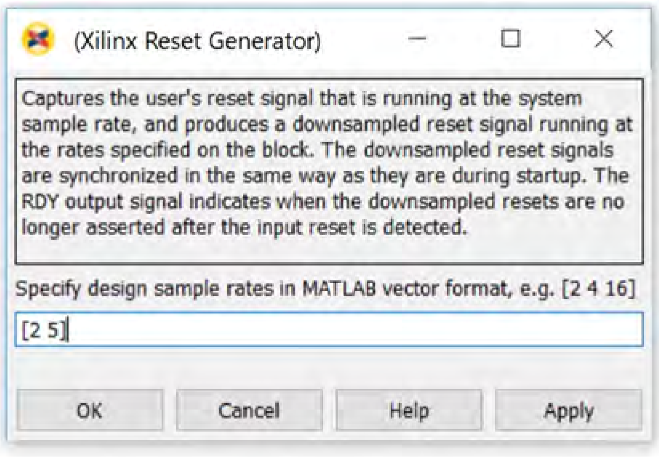


The downsampled reset signals are synchronized in the same way as they are during startup. The RDY output signal indicates when the downsampled resets are no longer asserted after the input reset is detected.

Block Parameters

The block parameters dialog box shown below can be invoked by double-clicking the icon in your Simulink® model.

Figure 364: **Block Parameters**



Captures the user's reset signal that is running at the system sample rate, and produces a downsampled reset signal running at the rates specified on the block. The downsampled reset signals are synchronized in the same way as they are during startup. The RDY output signal indicates when the downsampled resets are no longer asserted after the input reset is detected.

Specify design sample rates in MATLAB vector format, e.g. [2 4 16]

[2 5]

You specify the design sample rates in MATLAB® vector format as shown above. Any number of outputs can be specified.

ROM

The Xilinx ROM block is a single port read-only memory (ROM).

Values are stored by word and all words have the same arithmetic type, width, and binary point position. Each word is associated with exactly one address. An address can be any unsigned fixed-point integer from 0 to d-1, where d denotes the ROM depth (number of words). The memory contents are specified through a block parameter. The block has one input port for the memory address and one output port for data out. The address port must be an unsigned fixed-point integer. The block has two possible Xilinx LogiCORE™ implementations, using either distributed or block memory.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**

Parameters specific to the Basic tab are as follows.

- **Depth:** Specifies the number of words stored; must be a positive integer.
- **Initial value vector:** Specifies the initial value. When the vector is longer than the ROM depth, the vector's trailing elements are discarded. When the ROM is deeper than the vector length, the ROM's trailing words are set to zero. The initial value vector is saturated or rounded according to the data precision specified for the ROM.
- **Memory Type:** Specifies whether the ROM will be implemented using Distributed ROM or Block ROM. Depending on your selection, the ROM will be inferred or implemented as follows when the design is compiled:
  - If the block will be implemented in Distributed memory, the Distributed Memory Generator v8.0 LogiCORE IP will be inferred or implemented when the design is compiled. This is described in *Distributed Memory Generator LogiCORE IP Product Guide* (PG063).
  - If the block will be implemented in block RAM, the XPM_MEMORY_SPROM (Single Port ROM) macro will be inferred or implemented when the design is compiled. For information on the XPM_MEMORY_SPROM Xilinx Parameterized Macro (XPM), refer to *UltraScale Architecture Libraries Guide* (UG974).
- **Optional Ports:**
  - **Provide reset port for output register:** When selected, allows access to the reset port available on the output register of the Block ROM. The reset port is available only when the latency of the Block ROM is set to 1.
  - **Initial value for output register:** Specifies the initial value for output register. The initial value is saturated and rounded according to the data precision specified for the ROM.

- **Output tab:** Parameters specific to the Output tab are as follows.
  - **Output Type:** Specify the data type of the output.
    - **Boolean**
    - **Fixed-point**
    - **Floating-point**
  - **Arithmetic Type:** If the Output Type is specified as Fixed-point, you can select **Signed (2's comp)** or **Unsigned** as the Arithmetic Type.
  - **Fixed-point Precision:**
    - **Number of bits:** Specifies the bit location of the binary point of the output number, where bit zero is the least significant bit.
    - **Binary point:** Position of the binary point. in the fixed-point output.
  - **Floating-point Precision:**

    - **Single:** Specifies single precision (32 bits).
    - **Double:** Specifies double precision (64 bits).
    - **Custom:** Activates the field below so you can specify the Exponent width and the Fraction width.
- **Exponent width:** Specify the exponent width.
- **Fraction width:** Specify the fraction width.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

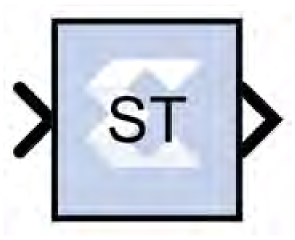
LogiCORE IP Distributed Memory Generator v8.0

UltraScale Architecture Libraries Guide - XPM_MEMORY_SPROM Macro (UltraRAM)

For the block memory, the address width must be equal to ceil(log2(d)) where d denotes the memory depth. The maximum width of data words in the block memory depends on the depth specified; the maximum depth is depends on the device family targeted. The tables below provide the maximum data word width for a given block memory depth.

Sample Time

The Sample Time block reports the normalized sample period of its input. A signal's normalized sample period is not equivalent to its Simulink absolute sample period. In hardware, this block is implemented as a constant.



Scalar2Vector

The Scalar2Vector block converts scalar type input to vector type output.

Description

The Xilinx® Scalar2Vector block slices the binary value of the input scalar based on the Width parameter to produce a vector output. For example, if the input is 3720 (binary value 111 010 001 000) of type `Ufix_12_0`, SSR is 4, and the Width parameter value is 3, then this block slices the input binary value into 4 groups, each of 3 bits, so that it produces the output [0 1 2 7].

Data Type Support

The input must be a Boolean or unsigned fixed-point signal. The output type is normally unsigned with binary point at zero, but can be Boolean when the Width parameter is 1.

Block Parameters

Width: This parameter defines the number of bits for each element of output vector. This parameter is decided by:

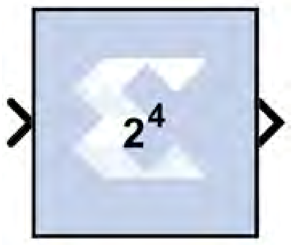
$$\text{Width} = \text{Input data width (bits)/SSR}$$

This formula must be satisfied when setting up the block parameters.

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control the processing of multiple data samples on every sample period. This block enables 1-D vector support for the primary block operation.

Scale

The Xilinx Scale block scales its input by a power of two. The power can be either positive or negative. The block has one input and one output. The scale operation has the effect of moving the binary point without changing the bits in the container



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

The only parameter that is specific to the Scale block is Scale factor s. It can be a positive or negative integer. The output of the block is i*2^k, where i is the input value and k is the scale factor. The effect of scaling is to move the binary point, which in hardware has no cost (a shift, on the other hand, might add logic).

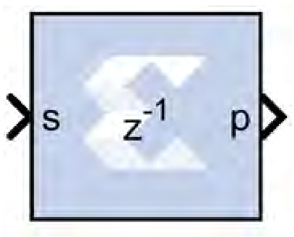
Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Xilinx LogiCORE

The Scale block does not use a Xilinx LogiCORE.

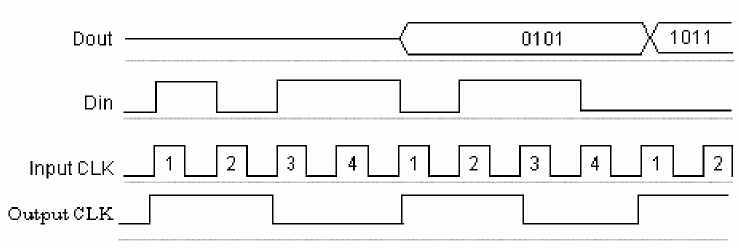
Serial to Parallel

The Serial to Parallel block takes a series of inputs of any size and creates a single output of a specified multiple of that size. The input series can be ordered either with the most significant word first or the least significant word first.



The following waveform illustrates the block's behavior:

Figure 365: **Serial to Parallel Behavior**



This example illustrates the case where the input width is 1, output width is 4, word size is 1 bit, and the block is configured for most significant word first.

Block Interface

The Serial to Parallel block has one input and one output port. The input port can be any size. The output port size is indicated on the block parameters dialog box.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

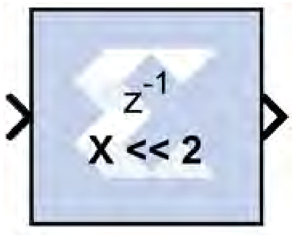
- **Basic tab:** Parameters specific to the Basic tab are as follows.
  - **Input order:** Least or most significant word first.
  - **Arithmetic type:** Signed or unsigned output.
  - **Number of bits:** Output width which must be a multiple of the number of input bits.
  - **Binary point:** Output binary point location

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

An error is reported when the number of output bits cannot be divided evenly by the number of input bits. The minimum latency for this block is zero.

Shift

The Xilinx Shift block performs a left or right shift on the input signal. The result will have the same fixed-point container as that of the input.



Block Parameters

Parameters specific to the Shift block are:

- **Shift direction**: specifies a direction, Left or Right. The Right shift moves the input toward the least significant bit within its container, with appropriate sign extension. Bits shifted out of the container are discarded. The Left shift moves the input toward the most significant bit within its container with zero padding of the least significant bits. Bits shifted out of the container are discarded.

- **Number of bits**: specifies how many bits are shifted. If the number is negative, direction selected with Shift direction is reversed.

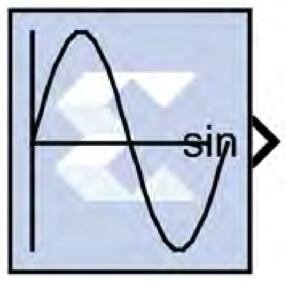
Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Xilinx LogiCORE

The Shift block does not use a Xilinx LogiCORE™.

Sine Wave

The Xilinx Sine Wave block generates a sine wave, using simulation time as the time source.



The Xilinx Sine Wave block outputs a sinusoidal waveform. Outputs from the block can be a sine wave, a cosine wave, or both. When implemented in a Xilinx FPGA or SoC, the Sine Wave block optimizes the block parameters for your target device.

The output of the Sine Wave block is determined by this equation:

$y = \sin(2\pi(k+o)/p)$

where

p = number of time samples per sine wave period

k = repeating integer value that ranges from 0 to p-1

o = offset (phase shift) of the signal

In this block, Model Composer sets *k* equal to 0 at the first time step and computes the block output, using the formula above. At the next time step, Simulink increments *k* and re-computes the output of the block. When *k* reaches *p*, Simulink resets *k* to 0 before computing the block output. This process continues until the end of the simulation.

The output characteristic of the Sine Wave block is determined by:

Samples per period = 2π / (Frequency * Sample Time)

Number of offset samples = Phase Offset * Samples per period / 2π

The Sine Wave block is ideal for generating simple sine and cosine waves. If your sine wave implementation will use more complicated features such as a phase generator, multiple channel support, or AXI4 ports, use the Xilinx DDS Compiler 6.0 block in your design instead of the Sine Wave block.

In the Vivado design flow, the Sine Wave block is inferred as "LogicCore IP DDS Compiler v6.0" for code generation.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

Parameters specific to the block are as follows:

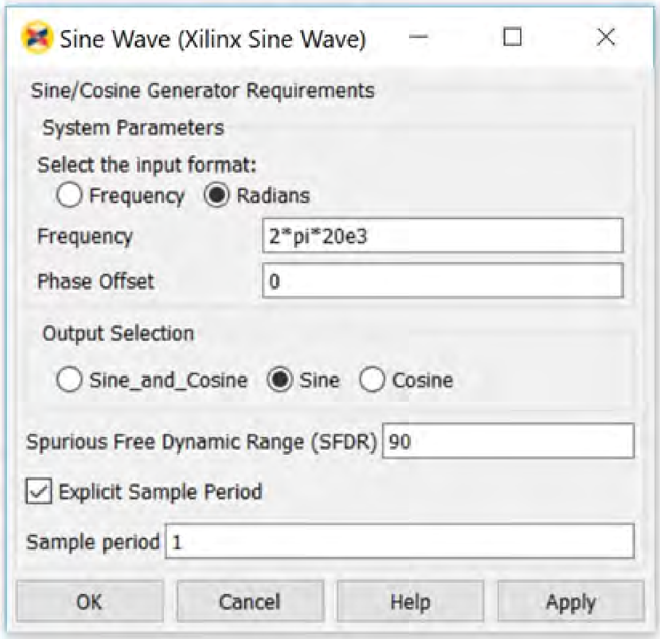
- **System Parameters:**
  - **Select the input format:** Specifies whether the frequency and phase offset inputs are entered as a **Frequency** (Hz) or an angular velocity (**Radians**) value.
  - **Frequency:** Specifies the frequency, either in Hertz or radians. The default is 1.
  - **Phase Offset:** Specifies the phase shift, either in Hertz or radians. The default is 0.
- **Output Selection:**
  - **Sine_and_Cosine:** Places both a sine and cosine output port on the block.
  - **Sine:** Places only a sine output port on the block.
  - **Cosine:** Places only a cosine output port on the block.
- **Spurious Free Dynamic Range (SFDR):** Specifies the precision of the output produced by the Sine Wave block. This sets the output width as well as internal bus widths, and controls various implementation decisions.
- **Explicit Sample Period:** If checked, the Sine Wave block uses the explicit sample time specified in the **Sample Period** box below. If not checked, the Model Composer base period will be used as block sample time.
- **Sample Period:** If **Explicit Sample Period** is selected, specifies the sample time for the block.

Example

A simple use case of generating sinusoidal signal using Sine Wave block is shown below.

To generate a 20 KHz sine wave with π/2 phase offset in a system running at sample period of (1/1e6) or 1 MHz, use the following specification on the Sine Wave block.

Figure 366: **Sine Wave Specifications**


Sine Wave (Xilinx Sine Wave)

Sine/Cosine Generator Requirements

System Parameters

Select the input format: Frequency [✓ Radians]

Frequency: 2*pi*20e3

Phase Offset: 0

Output Selection: Sine_and_Cosine [✓ Sine] Cosine

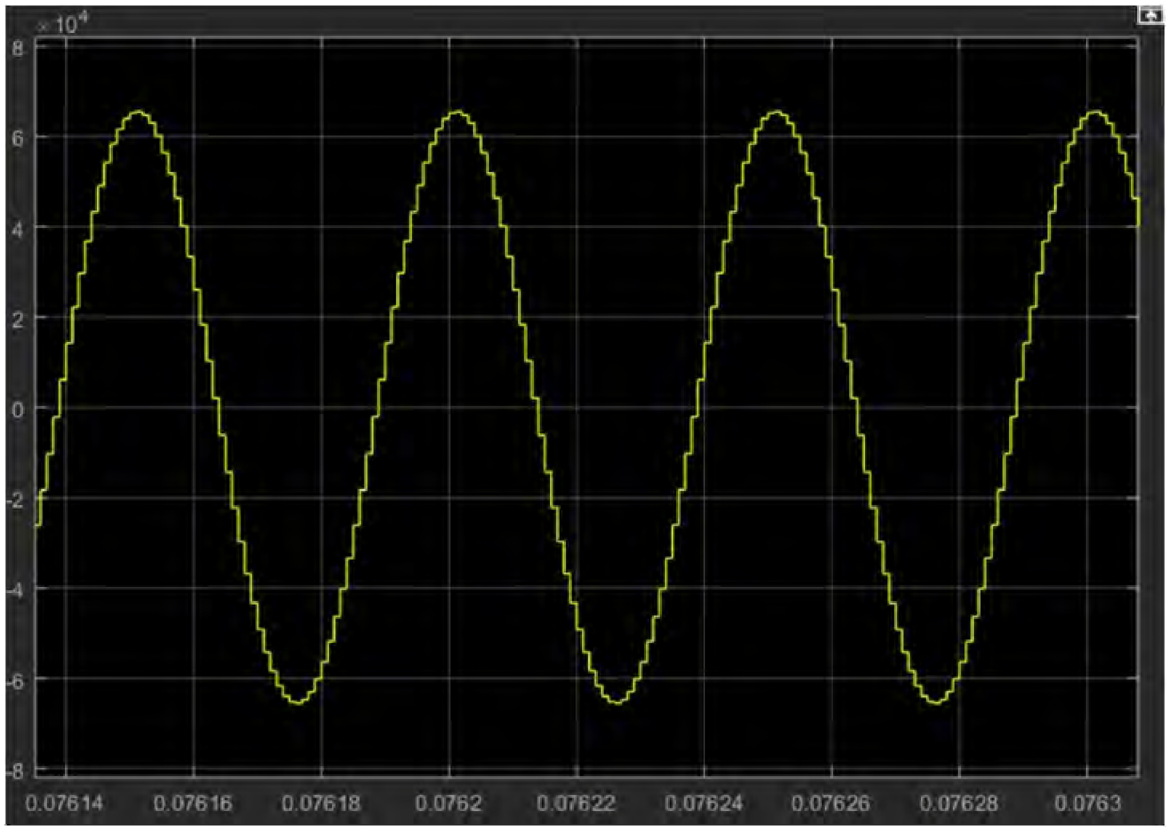
Spurious Free Dynamic Range (SFDR): 90

[✓] Explicit Sample Period

Sample period: 1

These settings generate this sine wave:

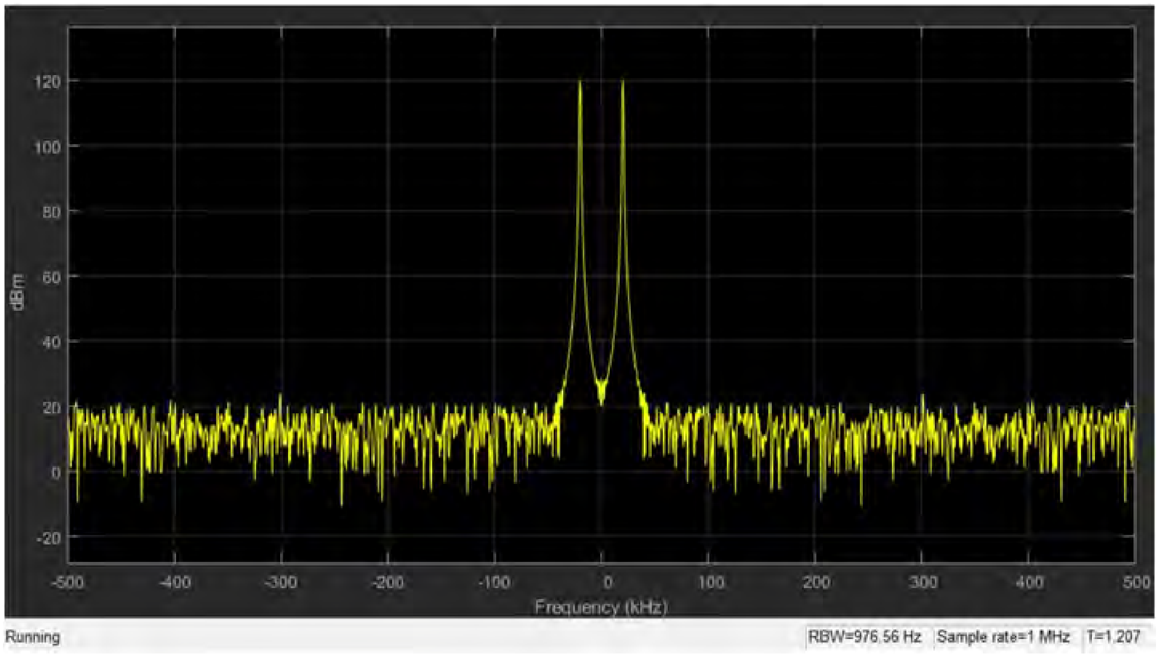
Figure 367: **Sine Wave Settings**



Wavelength of sine wave = *Simulink Sample Period / Frequency* => 1MHz/20KHz = 0.5 * 10^{-4}

The spectrum view of the sine wave is:

Figure 368: **Sine Wave Output**



Also:

Number of Samples per period = (2π/(1/1e6 * 20e3))

= 50 (Total number of samples in a single cycle)

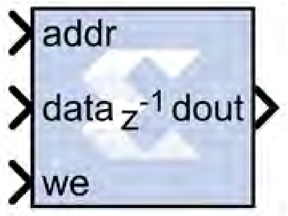
Number of offset samples = (π/2) * (50/2π) = 50/4

LogiCORE Documentation

LogiCORE IP DDS Compiler v6.0 Product Guide

Single Port RAM

The Xilinx Single Port RAM block implements a random access memory (RAM) with one data input and one data output port.



Block Interface

The block has one output port and three input ports for address, input data, and write enable (WE). Values in a Single Port RAM are stored by word, and all words have the same arithmetic type, width, and binary point position.

A single-port RAM can be implemented using either block memory, distributed memory, or UltraRAM resources in the FPGA. Each data word is associated with exactly one address that must be an unsigned integer in the range 0 to d-1, where d denotes the RAM depth (number of words in the RAM). An attempt to read past the end of the memory is caught as an error in the simulation, though if a block memory implementation is chosen, it can be possible to read beyond the specified address range in hardware (with unpredictable results). When the single-port RAM is implemented in distributed memory or block RAM, the initial RAM contents can be specified through the block parameters.

The write enable signal must be Bool, and when its value is 1, the data input is written to the memory location indicated by the address input. The output during a write operation depends on the choice of memory implementation.

The behavior of the output port depends on the write mode selected (see below). When the WE is 0, the output port has the value at the location specified by the address line.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic Tab:**

  Parameters specific to the Basic tab are as follows.

  - **Depth:** The number of words in the memory; must be a positive integer.
  - **Initial value vector:**

    The Initial value vector stores the initial contents of the memory. When the vector length exceeds the memory depth, values with index higher than depth are ignored. When the depth exceeds the vector length, memory locations with addresses higher than the vector length are initialized to zero. Initialization values are saturated and rounded (if necessary) according to the precision specified on the data port.

    UltraRAM memory is initialized to all 0's during power up or device reset. If implemented in UltraRAM, the Single Port RAM block cannot be initialized to user defined values.
  - **Memory Type:** Option to select whether the single-port RAM will be implemented using **Distributed memory**, **Block RAM**, or **UltraRAM**.

Depending on your selection for **Memory Type**, the single-port RAM will be inferred or implemented as follows when the design is compiled:

- If the block will be implemented in **Distributed memory**, the Distributed Memory Generator v8.0 LogiCORE IP will be inferred or implemented when the design is compiled. This LogiCORE IP is described in *Distributed Memory Generator LogiCORE IP Product Guide* (PG063).
- If the block will be implemented in block RAM or UltraRAM, the XPM_MEMORY_SPRAM (Single Port RAM) macro will be inferred or implemented when the design is compiled. For information on the XPM_MEMORY_SPRAM Xilinx Parameterized Macro (XPM), refer to *UltraScale Architecture Libraries Guide* (UG974).

- **Write Mode:** Specifies memory behavior when WE is asserted. Supported modes are: **Read after write**, **Read before write**, and **No read On write**. **Read after write** indicates the output value reflects the state of the memory after the write operation. **Read before write** indicates the output value reflects the state of the memory before the write operation. **No read on write** indicates that the output value remains unchanged irrespective of change of address or state of the memory. There are device specific restrictions on the applicability of these modes. Also refer to the Write Modes and Hardware Notes topics below for more information.
- **Provide reset port for output register:** For block RAM or UltraRAM, exposes a reset port controlling the output register of the RAM. This port does not reset the memory contents to the initialization value.

  ***Note*:** For Block RAM or UltraRAM, the reset port is available only when the latency of the Block RAM is greater than or equal to 1.
- **Initial value for output register:**

  for Block RAM, the initial value for the output register. The initial value is saturated and rounded as necessary according to the precision specified on the data port of the Block RAM.

  For UltraRAM, the output register is initialized to all 0's. The UltraRAM output register cannot be initialized to user defined values.

Other parameters used by this block are explained in the Common Options in Block Parameter Dialog Boxes topic at the beginning of this chapter.

Write Modes

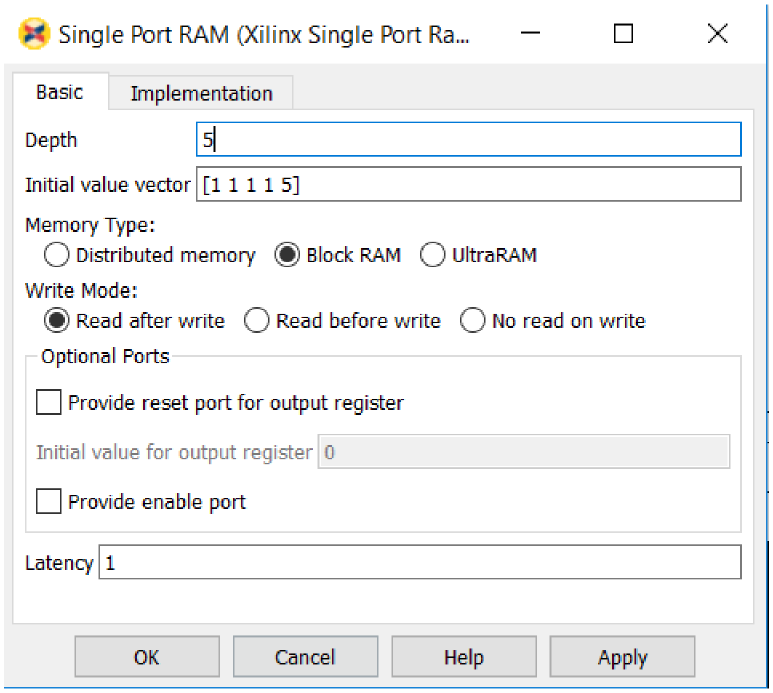
During a write operation (WE asserted), the data presented to the data input is stored in memory at the location selected by the address input. You can configure the behavior of the data out port A upon a write operation to one of the following modes:

- **Read after write**
- **Read before write**

- **No read on write**

These modes can be described with the help of the figure shown below. In the figure the memory has been set to an initial value of 5 and the address bit is specified as 4. When using **No read on write** mode, the output is unaffected by the address line and the output is the same as the last output when the WE was 0. For the other two modes, the output is obtained from the location specified by the address line, and hence is the value of the location being written to. This means that the output can be either the old value (**Read before write mode**), or the new value (**Read after write mode**).

Figure 369: **Configuration for Read After Write**



Depth: 5

Initial value vector: [1 1 1 1 5]

Memory Type: Distributed memory [✓ Block RAM] UltraRAM

Write Mode: [✓ Read after write] Read before write No read on write

Optional Ports

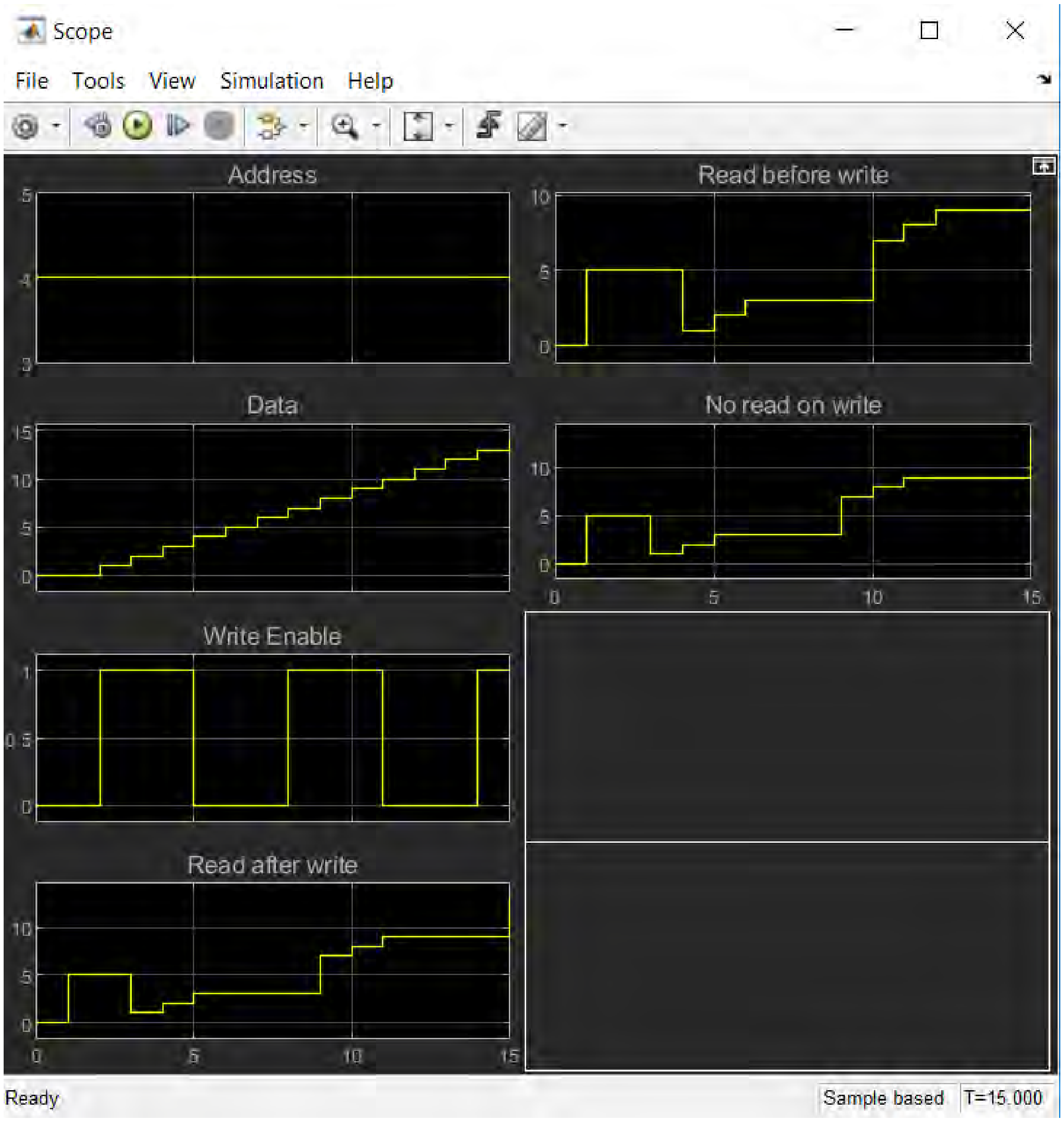
☐ Provide reset port for output register

Initial value for output register: 0

☐ Provide enable port

Latency: 1

Figure 370: **Write Output**



Hardware Notes

The distributed memory LogiCORE™ supports only the **Read before write** mode. The Xilinx Single Port RAM block also allows distributed memory with **Write Mode** option set to **Read after write** when specified latency is greater than 0. The **Read after write** mode for the distributed memory is achieved by using extra hardware resources (a MUX at the distributed memory output to latch data during a write operation).

LogiCORE and XPM Documentation

LogiCORE IP Distributed Memory Generator v8.0 (Distributed Memory)

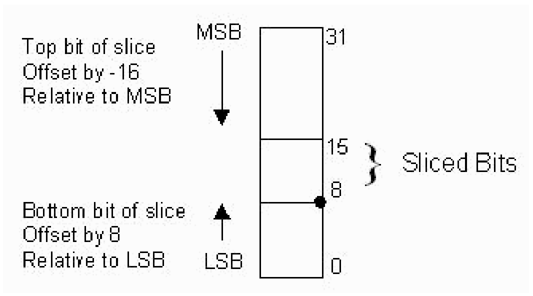
UltraScale Architecture Libraries Guide - XPM_MEMORY_SPRAM Macro (UltraRAM)

Slice

The Xilinx Slice block allows you to slice off a sequence of bits from your input data and create a new data value. This value is presented as the output from the block. The output data type is unsigned with its binary point at zero.

The block provides several mechanisms by which the sequence of bits can be specified. If the input type is known at the time of parameterization, the various mechanisms do not offer any gain in functionality. If, however, a Slice block is used in a design where the input data width or binary point position are subject to change, the variety of mechanisms becomes useful. The block can be configured, for example, always to extract only the top bit of the input, only the integral bits, or only the first three fractional bits. The following diagram illustrates how to extract all but the top 16 and bottom 8 bits of the input.

Figure 371: **Extracting Top 16 and Bottom 8 Bits**



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

Parameters specific to the block are as follows.

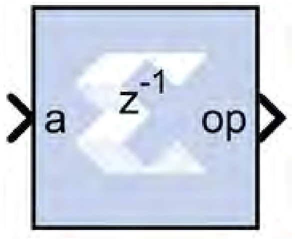
- **Width of slice (Number of bits):** Specifies the number of bits to extract.
- **Boolean output:** Tells whether single bit slices should be type Boolean.

- **Specify range as:** (Two bit locations | Upper bit location + width |Lower bit location + width). Allows you to specify either the bit locations of both end-points of the slice or one end-point along with number of bits to be taken in the slice.
- **Offset of top bit:** Specifies the offset for the ending bit position from the LSB, MSB or binary point.
- **Offset of bottom bit:** Specifies the offset for the ending bit position from the LSB, MSB or binary point.
- **Relative to:** Specifies the bit slice position relative to the Most Significant Bit (MSB), Least Significant Bit (LSB), or Binary point of the top or the bottom of the slice.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

SquareRoot

The Xilinx SquareRoot block performs the square root on the input. Currently, only the floating-point data type is supported.



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows.

  - **Flow Control:**
    - **Blocking:** Selects “Blocking” mode. In this mode, the lack of data on one input channel does block the execution of an operation if data is received on another input channel.
    - **NonBlocking:** Selects “Non-Blocking” mode. In this mode, the lack of data on one input channel does not block the execution of an operation if data is received on another input channel.
- **Optional ports:**

- **Input Channel Ports:**
  - **Has TLAST:** Adds a TLAST port to the Input channel.
  - **Has TUSER:** Adds a TUSER port to the Input channel.
  - **Provide enable port:** Adds an enable port to the block interface.
  - **Has Result TREADY:** Adds a TREADY port to the Result channel.
- **Exception Signals:**
  - **INVALID_OP:** Adds an output port that serves as an invalid operation flag.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP Floating-Point Operator v7.1

System Generator

The System Generator token serves as a control panel for controlling system and simulation parameters, and it is also used to invoke the code generator for netlisting. Every Simulink® model containing any element from the HDL Blockset must contain at least one System Generator token. Once a System Generator token is added to a model, it is possible to specify how code generation and simulation should be handled.

Token Parameters

The parameters dialog box can be invoked by double-clicking the icon in your Simulink model.

- **Compilation tab:**

  Parameters specific to the Compilation tab are as follows.

  - **Board:**

Specifies a Xilinx, Partner, or Custom board you will use to test your design. You can specify a **Board** for any of the compilation targets you select with the **Compilation** setting described below (**IP Catalog**, **Hardware Co-Simulation**, **Synthesized Checkpoint**, or **HDL Netlist**).

When you select a **Board**, the **Part** field displays the name of the Xilinx device on the selected **Board**, and this part name cannot be changed.

For a Partner board or a custom board to appear in the **Board** list, you must configure Model Composer to access the board files that describe the board.

- **Part:** Defines the Xilinx FPGA or SoC part to be used. If you have selected a **Board**, the **Part** field will display the name of the Xilinx device on the selected **Board**, and this part name cannot be changed.
- **Compilation:**

  Specifies the type of compilation result that should be produced when the code generator is invoked. The default compilation type is **IP Catalog**.

  The **Settings** button is activated when one of these compilation types is selected:

  - **IP Catalog compilation:** The **Settings** button brings up a dialog box that allows you to add a description of the IP that will be placed in the IP catalog.
  - **Hardware Co-Simulation (JTAG) compilation:** The **Settings** button brings up a dialog box that allows you to use burst data transfers to speed up JTAG hardware co-simulation.
- **Hardware Description Language:** Specifies the HDL language to be used for compilation of the design. The possibilities are VHDL and Verilog.
- **VHDL library:** Specifies the name of VHDL work library for code generation. The default name is **xil_defaultlib**.
- **Use STD_LOGIC type for Boolean or 1 bit wide gateways:** If your design's Hardware Description Language (HDL) is VHDL, selecting this option will declare a Boolean or 1-bit port (Gateway In or Gateway Out) as a STD-LOGIC type. If this option is not selected, Model Composer will interpret Boolean or 1-bit ports as vectors.

  ***Note*:** When you enable this option and try to run Generate code and Run behavioral simulation in Vivado, you may see a failure during the elaboration phase.
- **Target directory:** Defines where Model Composer should write compilation results. Because Model Composer and the FPGA physical design tools typically create many files, it is best to create a separate target directory, for example, a directory other than the directory containing your Simulink® model files.
- **Synthesis strategy:** Choose a Synthesis strategy from the pre-defined strategies in the drop-down list.

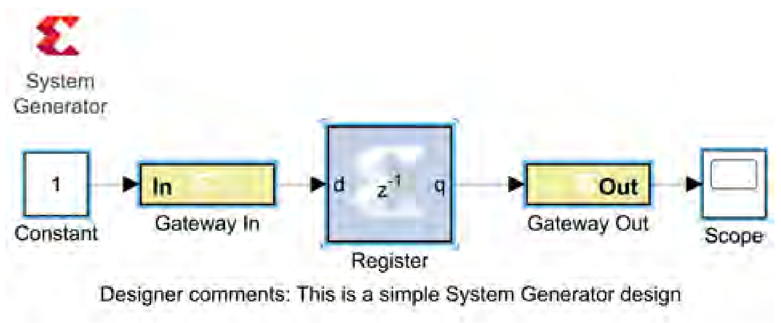
- **Implementation strategy:** Choose an Implementation strategy from the pre-defined strategies in the drop-down list.
- **Create interface document:**

  When this box is checked and the Generate button is activated for netlisting, Model Composer creates an HTM document that describes the design being netlisted. This document is placed in a “documentation” subfolder under the netlist folder.

  Adding Designer Comments to the Generated Document: If you want to add personalized comments to the auto-generated document, follow this procedure.

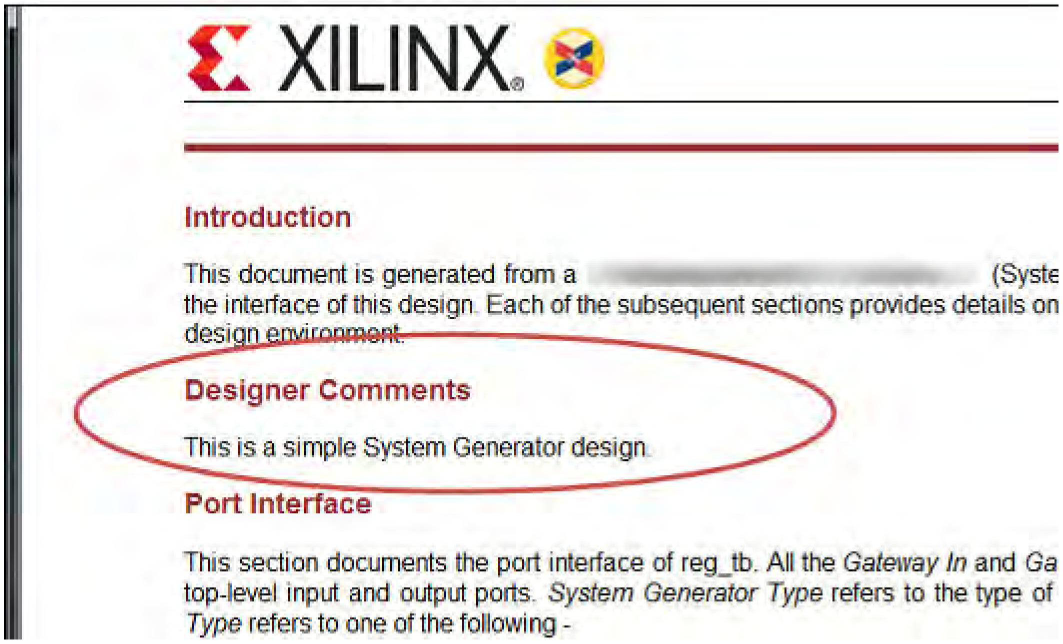
  1. As shown below, double click the Simulink canvas at the top level and add a comment that starts with **Designer Comments:**

     Figure 372: **Designer Comments**

  2. Double click on the System Generator token, click the **Create interface document** box at the bottom of the Compilation tab, then click **Generate**.
  3. When netlisting is complete, navigate to the documentation subfolder underneath the netlist folder and double click on the HTM document. As shown below,
  4. Designer Comments section is created in the document and your personalized comments are included.

Figure 373: **Designer Comments Section**



Introduction

This document is generated from a (Syste
the interface of this design. Each of the subsequent sections provides details on
design environment.

Designer Comments

This is a simple System Generator design.

Port Interface

This section documents the port interface of reg_tb. All the *Gateway In* and *Ga*
top-level input and output ports. *System Generator Type* refers to the type of
Type refers to one of the following -

- **Create testbench:** This instructs Model Composer to create an HDL test bench. Simulating the test bench in an HDL simulator compares Simulink simulation results with ones obtained from the compiled version of the design. To construct test vectors, Model Composer simulates the design in Simulink, and saves the values seen at gateways. The top HDL file for the test bench is named <name>_testbench.vhd/.v, where <name> is a name derived from the portion of the design being tested.

  Note: Testbench generation is not supported for designs that have gateways (Gateway In or Gateway Out) configured as an AXI4-Lite Interface

- **Model Upgrade:** Generates a Status Report that helps you identify and upgrade blocks that are not the latest available.

- **Clocking tab:**

  Parameters specific to the Clocking tab are as follows.

  - **Enable multiple clocks:** Must be enabled in the top-level System Generator token of a multiple clock design. This indicates to the Code Generation engine that the clock information for the various Subsystems must be obtained from the System Generator tokens contained in those Subsystems. If not enabled, then the design will be treated as a single clock design where all the clock information is inherited from the top-level System Generator token.

  - **FPGA clock period(ns):** Defines the period in nanoseconds of the system clock. The value need not be an integer. The period is passed to the Xilinx implementation tools through a constraints file, where it is used as the global PERIOD constraint. Multicycle paths are constrained to integer multiples of this value.

- **Clock pin location:** Defines the pin location for the hardware clock. This information is passed to the Xilinx implementation tools through a constraints file. This option should not be specified if the Model Composer design is to be included as part of a larger HDL design.

- **Provide clock enable clear pin:** This instructs Model Composer to provide a `ce_clr` port on the top-level clock wrapper. The ce_clr signal is used to reset the clock enable generation logic. Capability to reset clock enable generation logic allows designs to have dynamic control for specifying the beginning of data path sampling.

- **Simulink system period (sec):** Defines the Simulink System Period, in units of seconds. The Simulink system period is the greatest common divisor of the sample periods that appear in the model. These sample periods are set explicitly in the block dialog boxes, inherited according to Simulink propagation rules, or implied by a hardware oversampling rate in blocks with this option. In the latter case, the implied sample time is in fact faster than the observable simulation sample time for the block in Simulink. In hardware, a block having an oversampling rate greater than one processes its inputs at a faster rate than the data. For example, a sequential multiplier block with an over-sampling rate of eight implies a (Simulink) sample period that is one eighth of the multiplier block's actual sample time in Simulink. This parameter can be modified only in a master block.

- **Perform analysis:** Specifies whether an analysis (timing or resource) will or will not be performed on the Model Composer design when it is compiled. If **None** is selected, no timing analysis or resource analysis will be performed. If **Post Synthesis** is selected, the analysis will be performed after the design has been synthesized in the Vivado® toolset. If **Post Implementation** is selected, the analysis will be performed after the design is implemented in the Vivado toolset.

- **Analyzer type:** Two selections are provided: **Timing** or **Resource**. After generation is completed, a Timing Analyzer table or Resource Analyzer table is launched.

- **Launch analyzer:** Launches the Timing Analyzer or Resource Analyzer table, depending on the selection of **Analyzer type**. This will only work if you already ran analysis on the Simulink model and haven't changed the Simulink model since the last run.

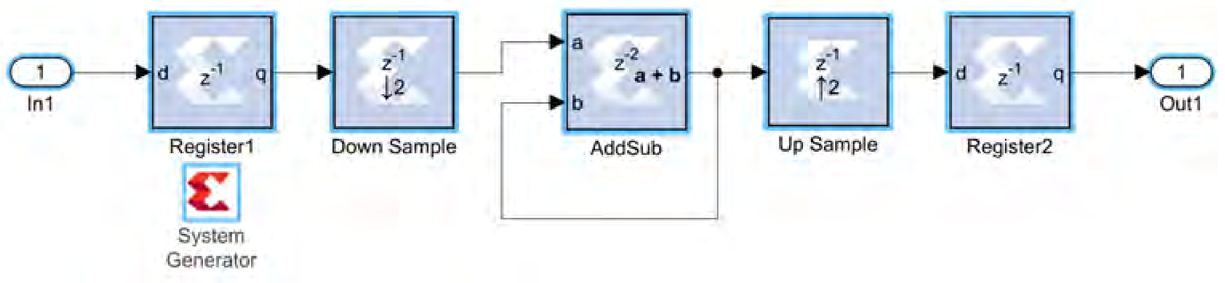
- **General tab:**

  Parameters specific to the General tab are as follows.

  - **Block icon display:** Specifies the type of information to be displayed on each block icon in the model after compilation is complete. The various display options are described below.

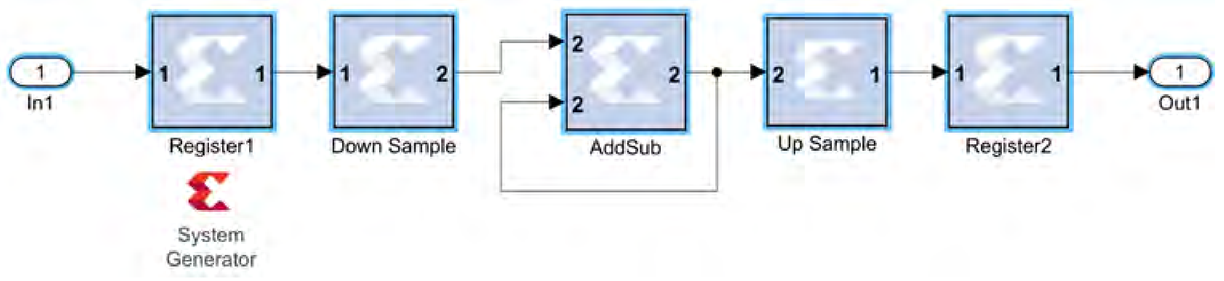
    - **Default:** Displays the default block icon information on each block in the model. A block's default icon is derived from the xbsIndex library.

Figure 374: **Default Block Icon**



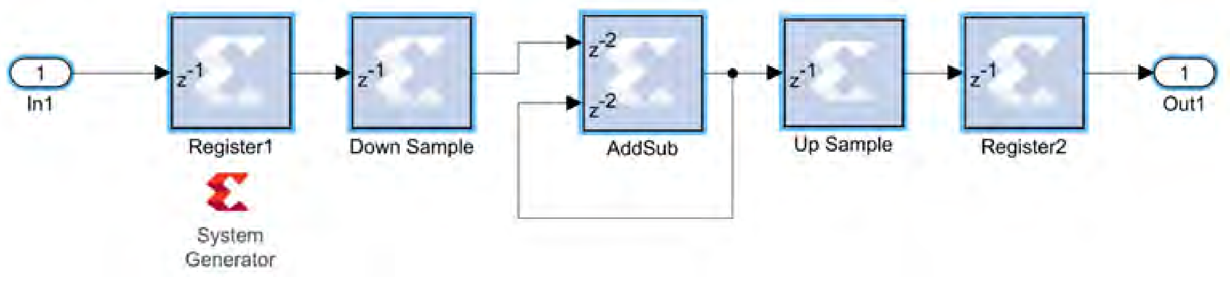
- **Normalized Sample Periods:** Displays the normalized sample periods for all the input and output ports on each block. For example, if the Simulink System Period is set to 4 and the sample period propagated to a block port is 4 then the normalized period that is displayed for the block port is 1 and if the period propagated to the block port is 8 then the sample period displayed would be 2 for example, a larger number indicates a slower rate.

Figure 375: **Normalized Sample Periods Icon**



- **Sample frequencies (MHz):** Displays sample frequencies for each block.
- **Pipeline stages:** Displays the latency information from the input ports of each block. The displayed pipeline stage might not be accurate for certain high-level blocks such as the FFT, RS Encoder/ Decoder, Viterbi Decoder, etc. In this case the displayed pipeline information can be used to determine whether a block has a combinational path from the input to the output. For example, the Up Sample block in the figure below shows that it has a combinational path from the input to the output port.

Figure 376: **Sample Frequencies**



    - **HDL port names:** Displays the HDL port name of each port on each block in the model.
    - **Input data types:** Displays the data type of each input port on each block in the model.
    - **Output data types:** Displays the data type of each output port on each block in the model.

- **Remote IP cache:**

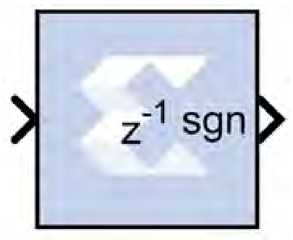
  If selected, your design will access an IP cache whenever a Model Composer compilation performs Vivado synthesis as part of the compilation. If the compilation generates an IP instance for synthesis, and the Vivado synthesis tool generates synthesis output products, the tools create an entry in the cache area. If a new customization of the IP is created which has the exact same properties, the tools will copy the synthesis outputs from the cache to the design's output directory instead of synthesizing the IP instance again. Accessing the disk cache speeds up the iterative design process.

  IP caching is described in HDL Library.

- **Clear cache:** Clicking this button clears the remote IP cache. Clearing the cache saves disk space, because the IP Cache can grow large, especially if your design uses many IP modules.

Threshold

The Xilinx Threshold block tests the sign of the input number. If the input number is negative, the output of the block is -1; otherwise, the output is 1. The output is a signed fixed-point integer that is 2 bits long. The block has one input and one output.



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

Parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

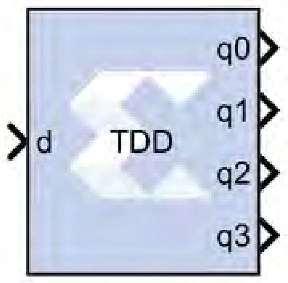
The block parameters do not control the output data type because the output is always a signed fixed-point integer that is 2 bits long.

Xilinx LogiCORE

The Threshold block does not use a Xilinx LogiCORE™.

Time Division Demultiplexer

The Xilinx Time Division Demultiplexer block accepts input serially and presents it to multiple outputs at a slower rate.

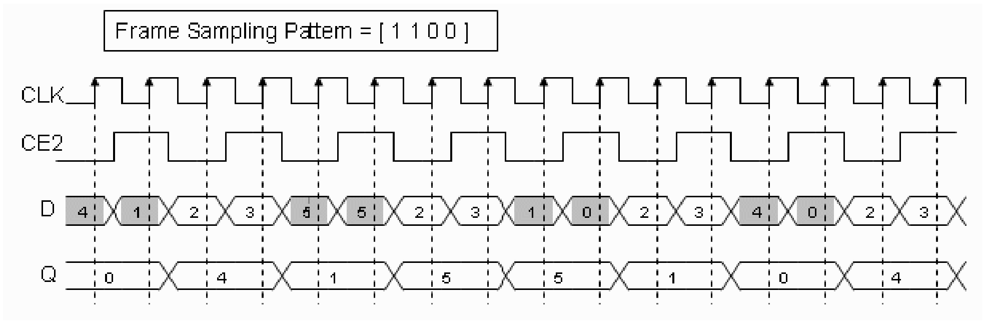


Block Interface

The block has one data input port and a user-configurable number of data outputs, ranging from 1 to 32. The data output ports have the same arithmetic type and precision as the input data port. The time division demultiplexer block also has optional input-valid port (vin) and output-valid port (vout). Both the valid ports are of type Bool.

For single channel implementation, the time division demultiplexer block has one data input and output port. Optional data valid input and output ports are also allowed. The length of the frame sampling pattern establishes the length of the input data frame. The position of 1 indicates the input value to be downsampled and the number of 1's correspond to the downsampling factor. The behavior of the demultiplexer block in single channel mode can best be illustrated with the help of the figure below. Based on the frame sampling pattern entered, the first and second input values of every input data frame are sampled and presented to the output at the rate of 2.

Figure 377: **Single Channel Implementation**



For single channel implementation, the number of values to be sampled from a data frame should evenly divide the size of the input frame. Every input data frame value can also be qualified by using the optional valid port.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

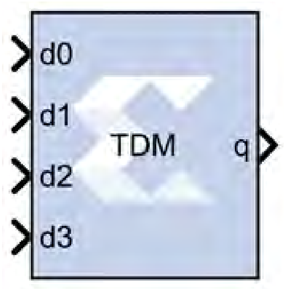
Parameters specific to this block are as follows.

- **Frame sampling pattern:** Specifies the size of the serial input data frame. The frame sampling pattern must be a MATLAB® vector containing only 1's and 0's.
- **Implementation:** Specifies the demultiplexer behavior to be either in single or multiple channel mode. The behaviors of these modes are explained above.
- **Provide Valid Port:** When selected, the demultiplexer has optional input and output valid ports (vin / vout). The vin port allows to qualify every input data value as part of the serial input data frame. The vout port marks the state of the output ports as valid or not.

Parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Time Division Multiplexer

The Xilinx Time Division Multiplexer block multiplexes values presented at input ports into a single faster rate output stream.



Block Interface

The block has two to 32 input ports and one output port. All input ports must have the same arithmetic type, precision, and rate. The output port has the same arithmetic type and precision as the inputs. The block has optional ports vin and vout that specify when input and output respectively are valid. Both valid ports are of type Bool.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

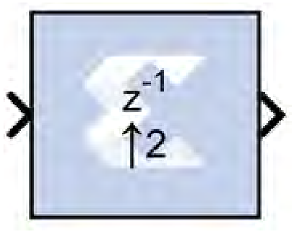
Parameters specific to the block are as follows.

- **Number of inputs:** Specifies the number of inputs (2 to 32).
- **Provide valid port:** When selected, the multiplexer is augmented with input and output valid ports named vin and vout respectively. When the vin port indicates that input values are invalid, the vout port indicates the corresponding output frame is invalid.
- **Optimization Parameter:** The Time Division Multiplexer block logic can be implemented in fabric (optimizing for resource usage) or in DSP48E1/DSP48E2 primitives (optimizing for speed). The default is **Resource**.
- **Resource:** Use combinatorial fabric (general interconnect) to implement the Time Division Multiplexer in the Xilinx device.
- **Speed:** Use DSP48 primitives to implement the Time Division Multiplexer in the Xilinx device.

Parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Up Sample

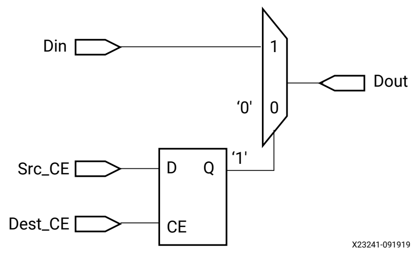
The Xilinx Up Sample block increases the sample rate at the point where the block is placed in your design. The output sample period is l/n, where l is the input sample period, and n is the sampling rate.



The input signal is up sampled so that within an input sample frame, an input sample is either presented at the output n times if samples are copied, or presented once with (n-1) zeroes interspersed if zero padding is used.

In hardware, the Up Sample block has two possible implementations. If the Copy Samples option is selected on the block parameters dialog box, the Din port is connected directly to Dout and no hardware is expended. Alternatively, if zero padding is selected, a mux is used to switch between the input sample and inserted zeros. The corresponding circuit for the zero padding Up Sample block is shown below.

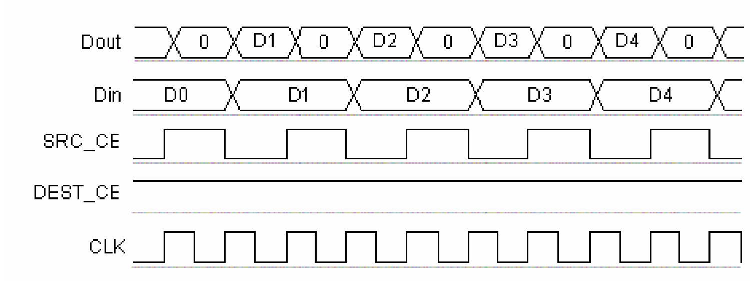
Figure 378: **Zero Padding Up Sample Circuit**



Block Interface

The Up Sample block receives two clock enable signals, Src_CE and Dest_CE. Src_CE is the clock enable signal corresponding to the input data stream rate. Dest_CE is the faster clock enable, corresponding to the output data stream rate. Notice that the circuit uses a single flip-flop in addition to the mux. The flip-flop is used to adjust the timing of Src_CE, so that the mux switches to the data input sample at the start of the input sample period, and switches to the constant zero after the first input sample. It is important to notice that the circuit has a combinational path from Din to Dout. As a result, an Up Sample block configured to zero pad should be followed by a register whenever possible.

Figure 379: **Up Sample Output**



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows.

  - **Sampling rate (number of output samples per input sample):** Must be an integer with a value of 2 or greater. This is the ratio of the output sample period to the input, and is essentially a sample rate multiplier. For example, a ratio of 2 indicates a doubling of the input sample rate. If a non-integer ratio is desired, the Up Sample block can be used in combination with the Down Sample block.

- **Copy samples (otherwise zeros are inserted):** Allows you to choose what to do with the additional samples produced by the increased clock rate. By selecting Copy Samples, the same sample is duplicated (copied) during the extra sample times. If this checkbox is not selected, the additional samples are zero.
- **Provide enable port:** When checked, this option adds an en (enable) input port, if the Latency is specified as a positive integer greater than zero.
- **Latency:** This defines the number of sample periods by which the block's output is delayed. One sample period can correspond to multiple clock cycles in the corresponding FPGA implementation (for example, when the hardware is over-clocked with respect to the Simulink® model). The user defined sample latency is handled in the Upsample block by placing shift registers that are clock enabled at the input sample rate, on the input of the block. The behavior of an Upsample block with non-zero latency is similar to putting a delay block, with equivalent latency, at the input of an Upsample block with zero latency.

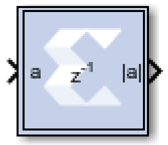
Parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Vector Absolute

The Vector Absolute block outputs the absolute value of the input of vector type.

Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control the processing of multiple data samples on each sample period. This block enables 1-D vector support for the primary block operation.



Block Parameters

Double-click the icon in your Simulink® model to open the Block Parameters dialog box.

- **Basic tab :**
  - **Precision:**

    This parameter allows you to specify the output precision for fixed-point arithmetic. Floating-point arithmetic output will always be **Full** precision.

    - **Full:** The block uses sufficient precision to represent the result without error.

- **User Defined:** If you do not need full precision, this option allows you to specify a reduced number of total bits and/or fractional bits.

- **Fixed-point Output Type:**

  Arithmetic type:

  - **Signed (2's comp):** The output is a Signed (2's complement) number.
  - **Unsigned:** The output is an Unsigned number.

- **Fixed-point Precision:**

  - **Number of bits:** Specifies the bit location of the binary point of the output number, where bit zero is the least significant bit.
  - **Binary point:** Position of the binary point. in the fixed-point output.

- **Quantization:**

  Refer to the section Overflow and Quantization.

- **Overflow:**

  Refer to the section Overflow and Quantization.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

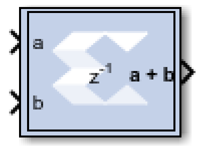
LogiCORE IP Floating-Point Operator v7.1

Vector AddSub Fabric

The Vector Adder/Subtracter Fabric block supports the Addition/Subtraction operation for inputs of vector type.

Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on each sample period. This blocks enable 1-D vector support for the primary block operation.



Block Parameters

Double-click the icon in your Simulink® model to open the block parameters dialog box.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows:

  - **Operation:** Specifies the block operation to be Addition, Subtraction, or Addition/Subtraction. When Addition/Subtraction is selected, the block operation is determined by the sub input port, which must be driven by a Boolean signal. When the sub input is 1, the block performs subtraction. Otherwise, it performs addition.
  - **Optional Ports:**
    - **Provide carry-in port:** When selected, allows access to the carry-in port, `cin`. The carry-in port is available only when **User defined** precision is selected and the binary point of the inputs is set to zero.
    - **Provide carry-out port:** When selected, allows access to the carry-out port, `cout`. The carry-out port is available only when **User defined** precision is selected, the inputs and output are unsigned, and the number of output integer bits equals x, where x = max (integer bits `a`, integer bits `b`).
  - **Latency:** The **Latency** value defines the number of sample periods by which the block's output is delayed. One sample period might correspond to multiple clock cycles in the corresponding FPGA implementation (for example, when the hardware is over-clocked with respect to the Simulink model). Model Composer does not perform extensive pipelining unless you select the **Pipeline for maximum performance** option (on the Implementation tab, described below); additional latency is usually implemented as a shift register on the output of the block.
- **Output tab:**
  - **Precision:**

    This parameter allows you to specify the output precision for fixed-point arithmetic. Floating point arithmetic output will always be **Full** precision.

    - **Full:** The block uses sufficient precision to represent the result without error.
    - **User Defined:** If you do not need full precision, this option allows you to specify a reduced number of total bits and/or fractional bits.
  - **User-Defined Precision:**
    - **Fixed-point Precision:**
      - **Signed (2’s comp):** The output is a Signed (2’s complement) number.
      - **Unsigned:** The output is an Unsigned number.

    - **Number of bits:** Specifies the bit location of the binary point of the output number, where bit zero is the least significant bit.
    - **Binary point:** Position of the binary point. in the fixed-point output.
  - **Quantization:** Refer to the section Overflow and Quantization.
  - **Overflow:** Refer to the section Overflow and Quantization.
- **Implementation tab:**

  Parameters specific to the Implementation tab are as follows:

  - **Use behavioral HDL (otherwise use core):** The block is implemented using behavioral HDL. This gives the downstream logic synthesis tool maximum freedom to optimize for performance or area.

    ***Note*:** For Floating-point operations, the block always uses the Floating-point Operator core.

  - **Core Parameters:**
    - **Implement using:** Core logic can be implemented in **Fabric** or in a **DSP48**, if a DSP48 is available in the target device. The default is **Fabric**.
    - **Pipeline for maximum performance:**

      The XILINX LogiCORE™ can be internally pipelined to optimize for speed instead of area. Selecting this option puts all user defined latency into the core until the maximum allowable latency is reached. If the **Pipeline for maximum performance** option is *not* selected and latency is greater than zero, a single output register is put in the core and additional latency is added on the output of the core.

      The **Pipeline for maximum performance** option adds the pipeline registers throughout the block, so that the latency is distributed, instead of adding it only at the end. This helps to meet tight timing constraints in the design.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

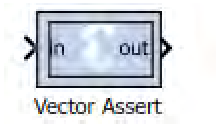
LogiCORE Documentation

LogiCORE IP Adder/Subtractor v12.0

LogiCORE IP Floating-Point Operator v7.1

Vector Assert

The Vector Assert block asserts a user-defined sample rate and/or type on Vector inputs.



Hardware notes: In hardware this blocks costs nothing.

Description

Super Sample Rate (SSR): Use this configurable GUI parameter to control processing of multiple data samples on every sample period. This block enables 1-D vector data support for the primary block operation.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

Parameters specific to this block are as follows:

- **Type:**
  - **Assert type:** Specifies whether or not the block will assert that the type at its input is the same as the type specified. If the types are not the same, an error message is reported.
  - **Specify type:** Specifies whether or not the type to assert is provided from a signal connected to an input port named type or whether it is specified **Explicitly** from parameters in the Assert block dialog box.
  - **Output Type:** Specifies the data type of the output. Can be **Boolean**, **Fixed-point**, or **Floating-point**.
  - **Arithmetic Type:** If the Output Type is specified as Fixed-point, you can select **Signed (2's comp)** or **Unsigned** as the Arithmetic Type.
    - **Fixed-point Precision:**
      - **Number of bits:** Specifies the bit location of the binary point of the output number, where bit zero is the least significant bit.
      - **Binary point:** Position of the binary point in the fixed-point output.
    - **Floating-point Precision:**
      - **Single:** Specifies single precision (32 bits).
      - **Double:** Specifies double precision (64 bits).
      - **Custom:** This block is listed in the following: Activates the field below so you can specify the Exponent width and the Fraction width.

- **Exponent width:** Specify the exponent width.
- **Fraction width:** Specify the fraction width.

- **Rate:**
  - **Assert rate:** specifies whether or not the block will assert that the rate at its input is the same as the rate specified. If the rates are not the same, an error message is reported.
  - **Specify rate:** Specifies whether or not the initial rate to assert is provided from a signal connected to an input port named `rate`, or whether it is specified **Explicitly** from the **Sample rate** parameter in the Assert block dialog box.
- **Provide output port:** Specifies whether or not the block will feature an output port. The type and/or rate of the signal presented on the output port is the type and/or rate specified for assertion.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

The **Output type** parameter in this block uses the same description as the Arithmetic Type described in the topic Common Options in Block Parameter Dialog Boxes.

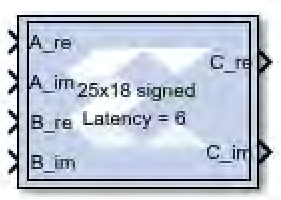
The Vector Assert block does not use a Xilinx LogiCORE™ and does not use resources when implemented in hardware.

Vector Complex Mult

The Vector Complex Multiplier block supports multiplication of two complex input vectors.

Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This blocks enable 1-D vector support for the primary block operation.



Data Type Support

- Supports fixed and floating-point data type inputs on both port A and B.
- The number of bits on Input port A should be greater than or equal to 26.

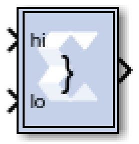
- The number of bits on Input port B should be greater than or equal to 17.

Vector Concat

The Vector Concat block concatenates two or more inputs of type vector. The output is cast to an unsigned value with the binary point at zero.

Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This block enables 1-D vector data support for the primary block operation.



The Vector Reinterpret block provides capabilities that can extend the functionality of the Vector Concat block.

Block Interface

The block has *n* input ports, where *n* is a value between 2 and 1024, inclusively, and one output port. The first and last input ports are labeled `hi` and `low`, respectively. Input ports between these two ports are not labeled. The input to the `hi` port occupies the most significant bits of the output, and the input to the `lo` port occupies the least significant bits of the output.

Block Parameters

Double-click the icon in your Simulink® model to open the Block Parameter dialog box.

Parameters specific to this block are as follows:

- **Number of Inputs**: Specifies number of inputs, between 2 and 1024, inclusively, to concatenate together.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

The Vector Concat block does not use a Xilinx LogiCORE™.

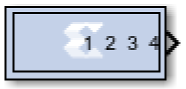
Vector Constant

The Vector Constant Block generates vector constant values.

Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This block enables 1-D vector data support for the primary block operation.

The Vector Constant block generates a constant that can be a fixed-point value, a Boolean value, or a DSP48 instruction. This block is similar to the Simulink® Vector Constant block, but can be used to directly drive the inputs on HDL blocks.



Block Parameters

Open the Block Parameters dialog box by double-clicking the icon in your Simulink model.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows:

  - **Constant Value:**

    Specifies the value of the constant. When changed, the new value appears on the block icon. If the constant data type is specified as fixed-point and cannot be expressed exactly in the specified fixed-point type, its value is rounded and saturated as needed. A positive value is implemented as an unsigned number, a negative value as signed.

  - **Output Type:** Specifies the data type of the output.
    - Boolean
    - Fixed-point
    - Floating-point
  - **Arithmetic Type:** If the Output Type is specified as Fixed-point.
    - Signed (2's comp)
    - Unsigned
    - DSP48 instruction
  - **Fixed-point Precision:**
    - **Number of bits:** Specifies the bit location of the binary point of the output number, where bit zero is the least significant bit.
    - **Binary point:** Position of the binary point. in the fixed-point output.

- **Floating-point Precision:**
  - **Single:** Specifies single precision (32 bits).
  - **Double:** Specifies double precision (64 bits).
  - **Custom:** Activates the field below so you can specify the Exponent width and the Fraction width.
  - **Exponent width:** Specifies the exponent width.
  - **Fraction width:** Specifies the fraction width.
- **Sample Period:**
  - **Sampled Constant:** Allows a sample period to be associated with the constant output and inherited by blocks that the constant block drives. (This is useful mainly because the blocks eventually target hardware and the Simulink sample periods are used to establish hardware clock periods.)

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes

Vector Convert

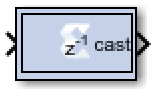
The Vector Convert block supports Data Type Conversion feature for vector type inputs.

Hardware notes: Rounding and saturating require hardware resources; truncating and wrapping do not.

Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This blocks enable 1-D vector data support for the primary block operation.

The Vector Convert block converts each input sample to a number of a desired arithmetic type. For example, a number can be converted to a signed (two's complement) or unsigned value.



Block Parameters

Open the Block Parameters dialog box by double-clicking the icon in your Simulink® model.

- **Basic tab:**

Parameters specific to the Basic Tab are as follows:

- **Output Type:**
    - Specify the output data type.
        - **Boolean**
        - **Fixed-point**
        - **Floating-point**
- **Arithmetic Type:** If the Output Type is specified as fixed-point, you can select Signed (two's comp) or Unsigned.
- **Fixed-point Precision:**
    - **Number of bits:** Specifies the bit location of the binary point, where bit zero is the least significant bit
    - **Binary point:** Specifies the bit location of the binary point, where bit zero is the least significant bit.
- **Floating-point Precision:**
    - **Single:** Specifies single precision (32 bits).
    - **Double:** Specifies double precision (64 bits).
    - **Custom:** Activates the field below so you can specify the Exponent width and the Fraction width.
    - **Exponent width:** Specify the exponent width.
    - **Fraction width:** Specify the fraction width.
- **Quantization:**

  Quantization errors occur when the number of fractional bits is insufficient to represent the fractional portion of a value. The options are to **Truncate** (for example, to discard bits to the right of the least significant representable bit), or to **Round(unbiased: +/- inf)** or **Round (unbiased: even values)**.

  Round(unbiased: +/- inf) also known as "Symmetric Round (towards +/- inf)" or "Symmetric Round (away from zero)". This is similar to the MATLAB `round()` function. This method rounds the value to the nearest desired bit away from zero and when there is a value at the midpoint between two possible rounded values, the one with the larger magnitude is selected. For example, to round 01.0110 to a Fix_4_2, this yields 01.10, since 01.0110 is exactly between 01.01 and 01.10 and the latter is further from zero.

Round (unbiased: even values) also known as "Convergent Round (toward even)" or "Unbiased Rounding". Symmetric rounding is biased because it rounds all ambiguous midpoints away from zero which means the average magnitude of the rounded results is larger than the average magnitude of the raw results. Convergent rounding removes this by alternating between a symmetric round toward zero and symmetric round away from zero. That is, midpoints are rounded toward the nearest even number. For example, to round 01.0110 to a Fix_4_2, this yields 01.10, since 01.0110 is exactly between 01.01 and 01.10 and the latter is even. To round 01.1010 to a Fix_4_2, this yields 01.10, since 01.1010 is exactly between 01.10 and 01.11 and the former is even.

- **Overflow:**

  Overflow errors occur when a value lies outside the representable range. For overflow the options are to **Saturate** to the largest positive/smallest negative value, to **Wrap** (for example, to discard bits to the left of the most significant representable bit), or to **Flag as error** (an overflow as a Simulink error) during simulation. **Flag as error** is a simulation only feature. The hardware generated is the same as when **Wrap** is selected.

- **Optional Ports:**

  Provide enable port: Activates an optional enable (en) pin on the block. When the enable signal is not asserted the block holds its current state until the enable signal is asserted again or the reset signal is asserted.

- **Latency:**

  The **Latency** value defines the number of sample periods by which the block's output is delayed. One sample period might correspond to multiple clock cycles in the corresponding FPGA implementation (for example, when the hardware is over-clocked with respect to the Simulink model). Model Composer will not perform extensive pipelining unless you select the **Pipeline for maximum performance** option (described below); additional latency is usually implemented as a shift register on the output of the block.

- **Implementation tab:**

  Parameters specific to the Implementation tab are as follows:

  - **Performance Parameters:**

    - **Pipeline for maximum performance:**

      The Xilinx LogiCORE™ can be internally pipelined to optimize for speed instead of area. Selecting this option puts all user defined latency into the core until the maximum allowable latency is reached. If the **Pipeline for maximum performance** option is *not* selected and latency is greater than zero, a single output register is put in the core and additional latency is added on the output of the core.

The **Pipeline for maximum performance** option adds the pipeline registers throughout the block, so that the latency is distributed, instead of adding it only at the end. This helps to meet tight timing constraints in the design.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP Floating-Point Operator v7.1

Vector DDFS

The Vector DDFS block generates Real and Imaginary vector output signals of desired frequency.

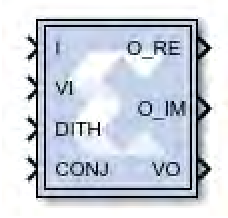
Description

The input port I is used for providing the desired output frequency value as a vector type.

The input port I value should be equal to:

(desired output Frequency * Sampling time *2^(Frequency Resolution))

The VI input port and VO output are used for synchronization purposes with the other blocks in the design. The VO is just a delayed version of VI based on the block latency. The DITH input port is used to turn on/off the phase noise dithering feature. The CONJ input port is set to '1' to conjugate the complex exponential output. The output ports, O_RE and O_IM, generate Real and Imaginary components of the desired vector output frequency signal.



Data Type Support

- The input port I should be a signed fixed-point data type.
- The input port VI, DITH, and CONJ should be Boolean data types.

Block Parameters

- **Super Sample Rate (SSR):**

This configurable GUI parameter is primarily used to control the processing of multiple data samples on every sample period. This block enables 1-D vector support for the primary block operation.

- **Frequency Resolution (bits):**

  Defines the smallest incremental step in frequency that the block can output. This should be an integer value.

- **Sin/Cos Table Depth:**

  Defines the depth of the Sin/Cos Table and should be an integer value.

- **Sin/Cos Table Width:**

  Defines the width of the Sin/Cos Table and should be an integer value.

Vector Delay

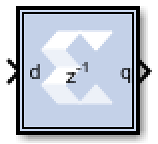
The Vector Delay block supports delay operation on vector type inputs.

Hardware notes: A delay line is a chain, each link of which is an SRL16 followed by a flip-flop.

Description

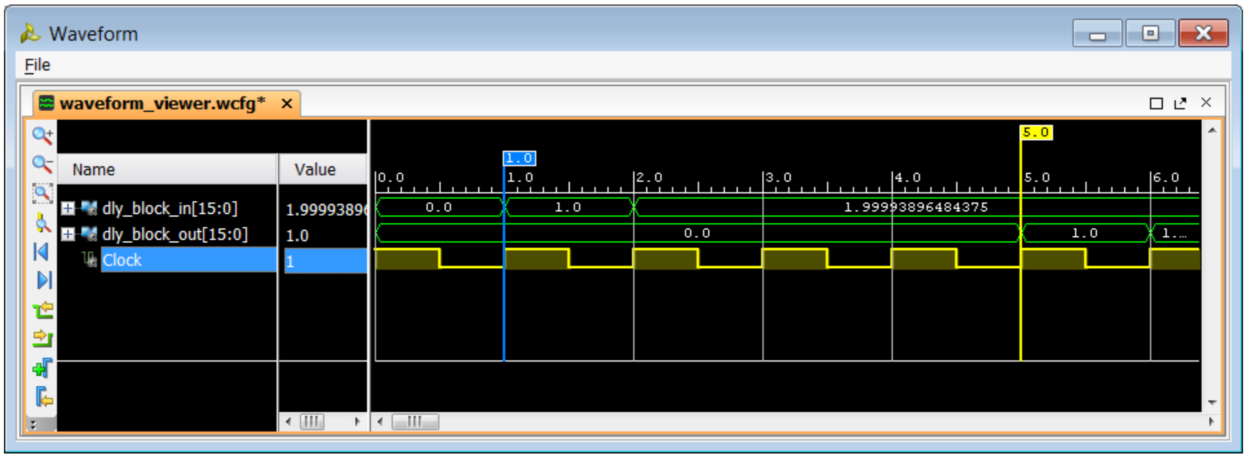
Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This block enables 1-D vector support for the primary block operation.

The Vector Delay block implements a fixed delay of L cycles.



The delay value is displayed on the block in the form z^{-L}, which is the *Z-transform* of the block's transfer function. Any data provided to the input of the block will appear at the output after L cycles. The rate and type of the data of the output is inherited from the input. This block is used mainly for matching pipeline delays in other portions of the circuit. The delay block differs from the register block in that the register allows a latency of only 1 cycle and contains an initial value parameter. The Vector Delay block supports a specified latency, but no initial value other than zeros. The figure below shows the Vector Delay block behavior when L=4 and Period=1s.

Figure 380: **Vector Delay block behavior when L=4 and Period=1s**



For delays that need to be adjusted during run-time, you should use the Addressable Shift Register block. Delays that are not an integer number of clock cycles are not supported and such delays should not be used in synchronous design (with a few rare exceptions).

Block Parameters

Open the Block Parameters dialog box by double-clicking the icon in your Simulink® model.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows:

  - **Optional Ports:**

    - **Provide synchronous reset port:** Activates an optional reset (rst) pin on the block. When the reset signal is asserted the block goes back to its initial state. Reset signal has precedence over the optional enable signal available on the block. The reset signal has to run at a multiple of the block's sample rate. The signal driving the reset port must be Boolean.

    - **Provide enable port:** Activates an optional enable (en) pin on the block. When the enable signal is not asserted the block holds its current state until the enable signal is asserted again or the reset signal is asserted. Reset signal has precedence over the enable signal. The enable signal has to run at a multiple of the block 's sample rate. The signal driving the enable port must be Boolean.

  - **Latency:** Latency is the number of cycles of delay. The latency can be zero, provided that the **Provide enable port** check box is not checked. The latency must be a non-negative integer. If the latency is zero, the Vector Delay block collapses to a wire during logic synthesis. If the latency is set to L=1, the block will generally be synthesized as a flip-flop (or multiple flip-flops if the data width is greater than 1).

- **Implementation tab:**

Parameters specific to the Implementation tab are as follows:

- **Implement using behavioral HDL**: Uses behavioral HDL as the implementation. This allows the downstream logic synthesis tool to choose the best implementation.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Logic Synthesis using Behavioral HDL

This setting is recommended if you are using Synplify Pro as the downstream logic synthesis tool. The logic synthesis tool will implement the delay as it desires, performing optimizations such as moving parts of the delay line back or forward into blockRAMs, DSP48s, or embedded IOB flip-flops; employing the dedicated SRL cascade outputs for long delay lines based on the architecture selected; and using flip-flops to terminate either or both ends of the delay line based on path delays. Using this setting also allows the logic synthesis tool, if sophisticated enough, to perform retiming by moving portions of the delay line back into combinational logic clouds.

Logic Synthesis using Structural HDL

If you do not check the **Implement using behavioral HDL** box, then structural HDL is used. This is the default setting and results in a known, but less-flexible, implementation which is often better for use with Vivado® synthesis. In general, this setting produces structural HDL comprising an SRL (Shift-Register LUT) delay of (L-1) cycles followed by a flip-flop, with the SRL and the flip-flop getting packed into the same slice. For a latency greater than L=33, multiple SRL/flip-flop sets are cascaded, albeit without using the dedicated cascade routes. For example, the following is the synthesis result for a 1-bit wide Vector Delay block with a latency of L=64.

Vector Delay Delta

The Vector Delay Delta Block delays each vector element differently based on the given latency and delay latency values.

Hardware notes: A delay line is a chain, each link of which is an SRL16 followed by a flip-flop.

Description

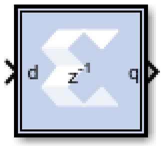
The delta latency parameter is used to generate each parallel path with different latency (for example, $[\text{Latency} + \text{Delta Latency} * (\text{i-1})]$, where *i* represents the channel number in a range from 1 to the SSR value).

The delta latency should be an integer and greater than or equal to $-\text{Latency}/(\text{SSR-1})$.

For example when SSR is set to '4', Latency is set to '1', and Delta Latency is set to '3' then the four channels from 1 to 4 are delayed by 1,4,7, and 10 sample times respectively.

Note: In the Vector Delay Delta block, all the parallel channels are delayed by an equal number of sample times provided by Latency parameter.

The Vector Delay Delta block implements a fixed delay of L cycles.



Block Parameters

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This blocks enable 1-D vector and/or complex data support for the primary block operation.

See the Xilinx® Vector Delay block for further information on using this block.

Vector Down Sample

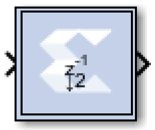
The Vector Down Sample block down samples input vector data.

Hardware notes: Sample and Latency controls determine the hardware implementation. The cost in hardware of different implementations varies considerably.

Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This blocks enable 1-D vector support for the primary block operation.

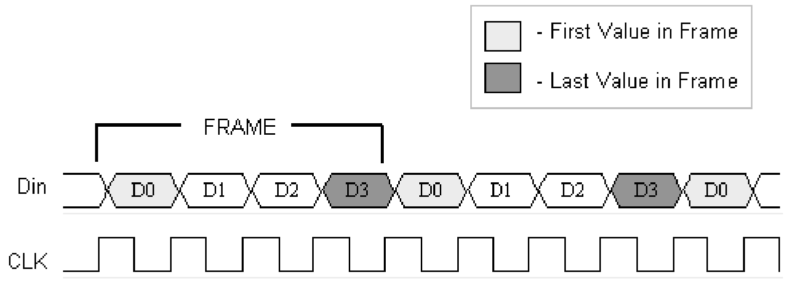
The Vector Down Sample block reduces the sample rate at the point where the block is placed in your design.



The input signal is sampled at even intervals, at either the beginning (first value), or end (last value) of a frame. The sampled value is presented on the output port and held until the next sample is taken.

A Vector Down Sample frame consists of l input samples, where l is sampling rate. An example frame for a Vector Down Sample block configured with a sampling rate of 4 is shown below.

Figure 381: **Vector Down Sample with Sampling Rate of 4**

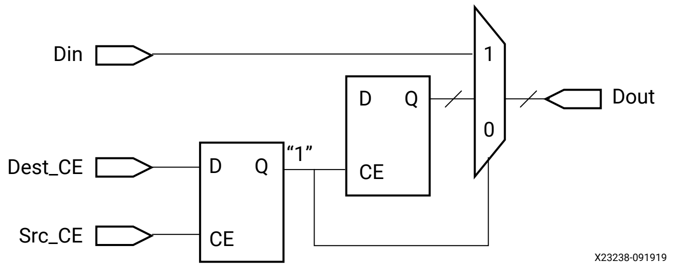


The Vector Down Sample block is realized in hardware using one of three possible implementations that vary in terms of implementation efficiency. The block receives two clock enable signals in hardware, `Src_CE`, and `Dest_CE`. `Src_CE` is the faster clock enable signal and corresponds to the input data stream rate. `Dest_CE` is the slower clock enable, corresponding to the output stream rate, for example, down sampled data. These enable signals control the register sampling in hardware.

Zero Latency Vector Down Sample

The zero latency Vector Down Sample block must be configured to sample the first value of the frame. The first sample in the input frame passes through the mux to the output port. A register samples this value during the first sample duration and the mux switches to the register output at the start of the second sample of the frame. The result is that the first sample in a frame is present on the output port for the entire frame duration. This is the least efficient hardware implementation as the mux introduces a combinational path from Din to Dout. A single bit register adjusts the timing of the destination clock enable, so that it is asserted at the start of the sample period, instead of the end. The hardware implementation is shown below.

Figure 382: **Zero Latency Vector Down Sample**

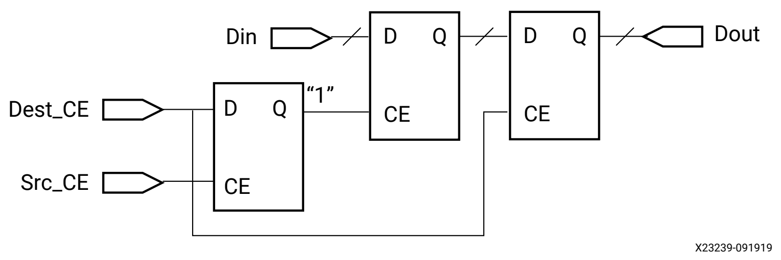


Vector Down Sample with Latency

If the Vector Down Sample block is configured with latency greater than zero, a more efficient implementation is used. One of two implementations is selected depending on whether the Vector Down Sample block is set to sample the first or last value in a frame.

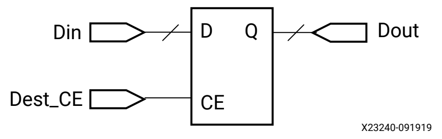
If the block samples the first value in a frame, two registers are required to correctly sample the input stream. The first register is enabled by the adjusted clock enable signal so that it samples the input at the start of the input frame. The second register samples the contents of the first register at the end of the sample period to ensure output data is aligned correctly.

Figure 383: **Two Register Example**



If the block samples the last value in a frame, a register samples the data input data at the end of the frame. The sampled value is presented for the duration of the next frame. The most efficient implementation is used when the Vector Down Sample block is configured to sample the last value of the frame.

Figure 384: **One Register Example**



Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

- **Basic tab:**

  Basic tab parameters are as follows.

  - **Sampling Rate (number of input samples per output sample):** Must be an integer greater or equal to 2. This is the ratio of the output sample period to the input, and is essentially a sample rate divider. For example, a ratio of 2 indicates a 2:1 division of the input sample rate. If a non-integer ratio is desired, the Vector Up Sample block can be used in combination with the Vector Down Sample block.
  - **Sample:** The Vector Down Sample block can sample either the first or last value of a frame. This parameter will determine which of these two values is sampled.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Xilinx LogiCORE

The Vector Down Sample block does not use a Xilinx® LogiCORE™.

Vector FFT

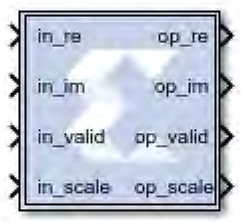
The Vector FFT block supports the FFT operation for vector type inputs.

Description

The real part of the input data should be given to the `in_re` port, and the imaginary part should be given to the `in_im port`.

When the `in_valid` is high it indicates that the input data is valid. When `out_valid` is high, it indicates that the output data is valid. The valid indicator accompanies every set of input and output samples (for example, R number of samples). There is no back pressure flow control and once an FFT transform starts, 'R' data samples must be input into the core every clock for N/R consecutive clocks. Where N is the FFT length. However, for back-to-back transforms, the valid control input can stay high with no gaps.

The `in_scale` input port is used if scaling is required. And `out_scale` is used in if there is an internal overflow.



Data Type Support

- The number of `in_scale` bits must be equal to `log2(FFT length)`.
- `in_valid` and `out_valid` are of Boolean data type.

Block Parameters

FFT length (N) is the size of the transformation, and should be powers of 2 in the range of 2^3 to 2^16. SSR is the super sample rate, the number of samples processed in parallel every clock. Using a typical example with N=1024 and SSR=4, the core would compute one 1K FFT every 256 clock cycles, processing 4 input samples/clock.

The fixed-point output data size must be 27 bits or less, this is limited by the DSP48 multiplier A port size.

BRAM_THRESHOLD is an implementation parameter with no functional implications, it controls the use of distributed RAM vs BRAM when implementing delay lines. It can be used to trade utilization numbers between these two types of resources. The higher the value, the more distributed RAM will be used instead of BRAM. Typical values to try are 258, 514, and 1026.

- **Scaling Ports:**

  The scaling ports are called SI and SO. Their width matches the FFT size N, it is always log2(N). There is one SI bit for every add/subtract stage, where internal overflows can occur. If that bit is set to zero then no scaling happens and bit growth is addressed by increasing the internal data sizes bit, one bit every stage. If the bit is set to 1 then the stage divides by 2, and no internal data growth is required to prevent overflows.

  Generally, if the output data size is log2(N) bits larger than the input size, no scaling is required, and SI is set to all zeros. If the input and output data sizes are equal, then scaling on every stage is needed and SI should be set to all ones. In reality, scaling is data dependent and some combination of output size growth and non-zero SI bits are used. If partial scaling is used, the non-zero bits of SI should be the MSB ones. SI should be static, it should not change while data is being processed by the core (when VI is high).

  Another important requirement to avoid internal overflows is to have one MSB margin bit at the data inputs, that is the two MSBs of I.RE and I.IM should be the same. This prevents overflows ink complex multipliers. If the two rules outlined above are followed, then internal overflows are impossible by design.

  The SO port is an indicator of internal overflows, it is not normally used, only attach an unsigned signals of size log2(N) to it.

Vector FIR

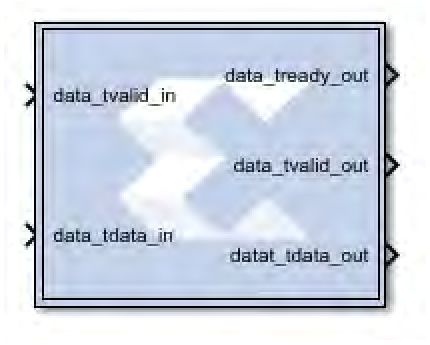
The Vector FIR block supports FIR filtering for vector type inputs.

Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This blocks enable 1-D vector support for the primary block operation.

- If the filter type is Interpolation, the output vector size (SSR value on the output side) is equal to the SSR value on the input side multiplied by Interpolation Rate Value.
- If the filter type is Decimation, the output vector size is equal to the SSR value on the input side divided by Decimation Rate Value.

This Vector FIR Compiler block provides a way to generate highly parameterizable, area-efficient, high-performance FIR filters with an AXI4-Stream-compliant interface.



AXI Ports that are Unique to this Block

This block exposes the AXI CONFIG channel as a group of separate ports based on sub-field names. The sub-field ports are described as follows:

Configuration Channel Input Signals:

config_tdata_fsel	A sub-field port that represents the **fsel** field in the Configuration Channel vector. **fsel** is used to select the active filter set. This port is exposed when the number of coefficient sets is greater than one. Refer to the FIR Compiler V7.2 Product Guide starting on page 5 for an explanation of the bits in this field.

Block Parameters

Open the block parameters dialog box by double-clicking the icon in your Simulink® model.

- **Filter Specification tab:**

  Parameters specific to the Filter Specification tab are as follows:

  - **Filter Coefficients:**

    - **Coefficient Vector:** Specifies the coefficient vector as a single MATLAB row vector. The number of taps is inferred from the length of the MATLAB row vector. If multiple coefficient sets are specified, then each set is appended to the previous set in the vector. It is possible to enter these coefficients using the FDATool block as well.

    - **Number of Coefficients Sets:** The number of sets of filter coefficients to be implemented. The value specified must divide without remainder into the number of coefficients.

    - **Use Reloadable Coefficients:** Check to add the coefficient reload ports to the block. The set of data loaded into the reload channel will not take action until triggered by a re-configuration synchronization event. Refer to the FIR Compiler V7.2 Product Guide for a more detailed explanation of the RELOAD Channel interface timing. This block supports the xlGetReloadOrder function. See the Model Composer Utility function xlGetReloadOrder for details.

  - **Filter Specification:**

- **Filter Type:**
  - **Single_Rate:** The data rate of the input and the output are the same.
  - **Interpolation:** The data rate of the output is faster than the input by a factor specified by the Interpolation Rate value.
  - **Decimation:** The data rate of the output is slower than the input by a factor specified in the Decimation Rate Value.
- **Rate Change Type:** This field is applicable to Interpolation and Decimation filter types. Used to specify an **Integer** or **Fixed_Fractional** rate change.
- **Interpolation Rate Value:** This field is applicable to all Interpolation filter types and Decimation filter types for Fractional Rate Change implementations. The value provided in this field defines the up-sampling factor, or P for Fixed Fractional Rate (P/Q) resampling filter implementations.
- **Decimation Rate Value:** This field is applicable to the all Decimation and Interpolation filter types for Fractional Rate Change implementations. The value provided in this field defines the down-sampling factor, or Q for Fixed Fractional Rate (P/Q) resampling filter implementations.
- **Zero pack factor:** Allows you to specify the number of 0's inserted between the coefficient specified by the coefficient vector. A zero packing factor of k inserts k-1 0s between the supplied coefficient values. This parameter is only active when the Filter type is set to Interpolated.
- **SSR:** SSR value.

- **Implementation tab:**

  Parameters specific to the Implementation tab are as follows:

  - **Coefficient Options:**
    - **Coefficient Type:** Specify Signed or Unsigned.
    - **Quantization:** Specifies the quantization method to be used for quantizing the coefficients. This can be set to one of the following:
      - Integer_Coefficients
      - Quantize_Only
      - Maximize_Dynamic_Range
      - Normalize_to_Centre_Coefficient
    - **Coefficient Width:** Specifies the number of bits used to represent the coefficients.
    - **Best Precision Fractional Bits:** When selected, the coefficient fractional width is automatically set to maximize the precision of the specified filter coefficients.

        - **Coefficient Fractional Bits:** Specifies the binary point location in the coefficients datapath options
        - **Coefficients Structure:**
            - Specifies the coefficient structure. Depending on the coefficient structure, optimizations are made in the core to reduce the amount of hardware required to implement a particular filter configuration. The selected structure can be any of the following:
                - Inferred
                - Non-Symmetric
                - Symmetric

          The vector of coefficients specified must match the structure specified unless Inferred from coefficients is selected in which case the structure is determined automatically from these coefficients.

    - **Datapath Options:**
        - **Output Rounding Mode:**
            - Choose one of the following:
                - Full_Precision
                - Truncate_LSBs
                - Non_Symmetric_Rounding_Down
                - Non_Symmetric_Rounding_Up
                - Symmetric_Rounding_to_Zero
                - Symmetric_Rounding_to_Infinity
                - Convergent_Rounding_to_Even
                - Convergent_Rounding_to_Odd
        - **Output Width:** Specify the output width. Edit box activated only if the Rounding mode is set to a value other than Full_Precision.

- **Detailed Implementation tab:**

  Parameters specific to the Detailed Implementation tab are as follows:

    - **Filter Architecture:**

      The following two filter architectures are supported.

        - Systolic_Multiply_Accumulate

- Transpose_Multiply_Accumulate

  Note: When selecting the Transpose Multiply-Accumulate architecture, these limitations apply:

  - Symmetry is not exploited. If the **Coefficient Vector** specified on the Filter Specification tab is detected as symmetric, the FIR Compiler 7.2 block parameters dialog box will not allow you to select Transpose Multiply Accumulate.
  - Multiple interleaved channels are not supported.

- **Optimization Options:** Specifies if the core is required to operate at maximum possible speed (“Speed” option) or minimum area (“Area” option). The “Area” option is the recommended default and will normally achieve the best speed and area for the design, however in certain configurations, the “Speed” setting might be required to improve performance at the expense of overall resource usage (this setting normally adds pipeline registers in critical paths).

  - **Goal:**
    - Area
    - Speed
    - Custom
  - **List:**

    A comma delimited list that specifies which optimizations are implemented by the block. The optimizations are as follows.

    - **Data_Path_Fanout:** Adds additional pipeline registers on the data memory outputs to minimize fan-out. Useful when implementing large data width filters requiring multiple DSP slices per multiply-add unit.
    - **Pre-Adder_Pipeline:** Pipelines the pre-adder when implemented using fabric resources. This may occur when a large coefficient width is specified.
    - **Coefficient_Fanout:** Adds additional pipeline registers on the coefficient memory outputs to minimize fan-out. Useful for Parallel channels or large coefficient width filters requiring multiple DSP slices per multiply-add unit.
    - **Control_Path_Fanout:** Adds additional pipeline registers to control logic when Parallel channels have been specified.
    - **Control_Column_Fanout:** Adds additional pipeline registers to control logic when multiple DSP columns are required to implement the filter.
    - **Control_Broadcast_Fanout:** Adds additional pipeline registers to control logic for fully parallel (one clock cycle per channel per input sample) symmetric filter implementations.
    - **Control_LUT_Pipeline:** Pipelines the Look-up tables required to implement the control logic for Advanced Channel sequences.

  - **No_BRAM_Read_First_Mode:** Specifies that Block RAM READ-FIRST mode should not be used.
  - **Increased speed:** Multiple DSP slice columns are required for non-symmetric filter implementations.
  - **Other:** Miscellaneous optimizations.

  ***Note*:** All optimizations maybe specified but are only implemented when relevant to the core configuration.

- **Memory Options:**

  The memory type for MAC implementations can either be user-selected or chosen automatically to suit the best implementation options. Note that a choice of "Distributed" might result in a shift register implementation where appropriate to the filter structure. Forcing the RAM selection to be either Block or Distributed should be used with caution, as inappropriate use can lead to inefficient resource usage - the default Automatic mode is recommended for most applications.

  - **Data Buffer Type:** Specifies the type of memory used to store data samples.
  - **Coefficient Buffer Type:** Specifies the type of memory used to store the coefficients.
  - **Input Buffer Type:** Specifies the type of memory to be used to implement the data input buffer, where present.
  - **Output Buffer type:** Specifies the type of memory to be used to implement the data output buffer, where present.
  - **Preference for other storage:** Specifies the type of memory to be used to implement general storage in the datapath.

- **DSP Slice Column Options:**

  - **Multi-Column Support:** For device families with DSP slices, implementations of large high speed filters might require chaining of DSP slice elements across multiple columns. Where applicable (the feature is only enabled for multi-column devices), you can select the method of folding the filter structure across the multiple-columns, which can be Automatic (based on the selected device for the project) or Custom (you select the length of the first and subsequent columns).
  - **Inter-Column Pipe Length:** Pipeline stages are required to connect between the columns, with the level of pipelining required being depending on the required system clock rate, the chosen device and other system-level parameters. The choice of this parameter is always left for you to specify.

- **Interface tab:**

  - **Data Channel Options:**

- **Output TREADY:** This field enables the data_tready port. With this port enabled, the block will support back-pressure. Without the port, back-pressure is not supported, but resources are saved and performance is likely to be higher.

- **Control Options:**
  - **ACLKEN:** Active-high clock enable. Available for MAC-based FIR implementations.
  - **ARESETn (active low):** Active-low synchronous clear input that always takes priority over ACLKEN. A minimum ARESETn active pulse of two cycles is required, since the signal is internally registered for performance. A pulse of one cycle resets the control and datapath of the core, but the response to the pulse is not in the cycle immediately following.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP FIR Compiler v7.2

Vector Logical

The Vector Logical block supports logical operation for vector type inputs.

Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control the processing of multiple data samples on every sample period. This blocks enable 1-D vector data support for the primary block operation.

The Vector Logical Block performs bitwise logical operations on fixed-point numbers. Operands are zero padded and sign extended as necessary to make binary point positions coincide. The logical operation is performed and the result is delivered at the output port.

in1
z⁻¹
AND
out
in2

In hardware this block is implemented as synthesizable VHDL. If you build a tree of logical gates, this synthesizable implementation is best as it facilitates logic collapsing in synthesis and mapping.

Block Parameters

Double-click the icon in your Simulink® model to open the Block Parameters dialog box.

- **Basic tab :** Parameters specific to the Basic tab are as follows:
  - **Logical function:** Specifies one of the following bitwise logical operators: AND, NAND, OR, NOR, XOR, XNOR.
  - **Number of inputs:** Specifies the number of inputs (1 - 1024).
  - **Logical Reduction Operation:**

    When the number of inputs is specified as 1, a unary logical reduction operation performs a bit-wise operation on the single operand to produce a single bit result. The first step of the operation applies the logical operator between the least significant bit of the operand and the next most significant bit. The second and subsequent steps apply the operator between the one-bit result of the prior step and the next bit of the operand using the same logical operator. The logical reduction operator implements the same functionality as that of the logical reduction operation in HDLs. The output of the logical reduction operation is always Boolean.
- **Output Type tab :**

  Parameters specific to the Output Type tab are as follows:
  - **Align binary point**: Specifies that the block must align binary points automatically. If not selected, all inputs must have the same binary point position.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

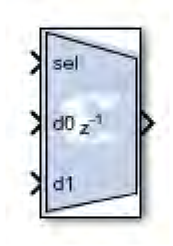
Vector Mux

The Vector Multiplexer block supports the Multiplexing feature for input of vector types.

Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This block enables 1-D vector support for the primary block operation.

The Vector Mux block implements a multiplexer. The block has one select input (type unsigned) and a user-configurable number of data bus inputs, ranging from 2 to 1024.



Block Parameters

Open the Block Parameters dialog box by double-clicking the icon in your Simulink® model.

- **Basic tab :**
  - **Number of inputs:** Specify a number between 2 and 32.
  - **Optional Ports:**

    Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.
- **Output tab :**
  - **Precision:**

    This parameter allows you to specify the output precision for fixed-point arithmetic. Floating-point arithmetic output will always be **Full** precision.

    - **Full:** The block uses sufficient precision to represent the result without error.
    - **User Defined:** If you do not need full precision, this option allows you to specify a reduced number of total bits and/or fractional bits.
  - **Fixed-point Output Type:**
    - **Arithmetic type:**
      - **Signed (2's comp):** The output is a Signed (2's complement) number.
      - **Unsigned:** The output is an Unsigned number.
    - **Fixed-Point Precision:**
      - **Number of bits:** Specifies the bit location of the binary point of the output number where bit zero is the least significant bit.
      - **Binary point:** Position of the binary point. in the fixed-point output.
    - **Quantization:**

Refer to the section Overflow and Quantization in the topic Common Options in Block Parameter Dialog Boxes.

- **Overflow:**

  Refer to the section Overflow and Quantization in the topic Common Options in Block Parameter Dialog Boxes.

Parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP Floating-Point Operator v7.1

Vector Real Gateway In

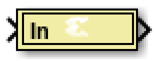
The Vector Real Gateway In block converts vector inputs of type Simulink® integer, single, double, and fixed-point to Xilinx® fixed-point or floating-point data type.

Hardware notes: In hardware, these blocks become top level input ports.

Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This block enables 1-D vector support for the primary block operation.

The Vector Real Gateway In blocks are the inputs into the HDL portion of your Simulink design. These blocks convert Simulink integer, double and fixed-point data types into the Model Composer fixed-point type. Each block defines a top-level input port or interface in the HDL design generated by Model Composer.



Conversion of Simulink Data to Model Composer Data

A number of different Simulink data types are supported on the input of Vector Real Gateway In. The data types supported include int8, uint8, int16, uint16, in32, uint32, single, double, and Simulink fixed-point data type(if Simulink fixed-point data type license is available). In all causes the input data is converted to a double internal to gateway, and then converted to target data type as specified on the Vector Real Gateway In block (Fixed-point, Floating-point or Boolean). When converting to Fixed-point from the internal double representation, the Quantization, and Overflow is further handled as specified in the Block GUI. For overflow, the options are to saturate to the largest positive/smallest negative value, to wrap (for example, to discard bits to the left of the most significant representable bit), or to flag an overflow as a Simulink error during simulation. For quantization, the options are to round to the nearest representable value (or to the value furthest from zero if there are two equidistant nearest representable values), or to truncate (for example, to discard bits to the right of the least significant representable bit). It is important to realize that conversion, overflow and quantization do not take place in hardware. They take place only in the simulation model of the block.

Gateway Blocks

As listed below, the Xilinx Vector Real Gateway In block is used to provide a number of functions:

- Converting data from Simulink integer, double, and fixed-point type to the Model Composer fixed-point type during simulation in Simulink.
- Defining top-level input ports or interface in the HDL design generated by Model Composer.
- Defining test bench stimuli when the **Create Testbench** box is checked in the System Generator token. In this case, during HDL code generation, the inputs to the block that occur during Simulink simulation are logged as a logic vector in a data file. During HDL simulation, an entity that is inserted in the top level test bench checks this vector, and the corresponding vectors produced by Vector Real Gateway Out blocks against expected results.
- Naming the corresponding port in the top level HDL entity.

Block Parameters

Open the Block Parameters dialog box double-clicking the icon in your Simulink model.

- **Basic Tab :**

  Parameters specific to the Basic Tab are as follows:

  - **Output Type:**

    Specifies the output data type. Can be **Boolean**, **Fixed-point**, or **Floating-point**.

- **Arithmetic Type:**

  If the Output Type is specified as Fixed-point, you can select **Signed (2's comp)** or **Unsigned** as the Arithmetic Type.

- **Fixed-point Precision:**

  - **Number of bits:** Specifies the bit location of the binary point, where bit zero is the least significant bit.

  - **Binary point:** Specifies the bit location of the binary point, where bit zero is the least significant bit.

- **Floating-point Precision:**

    - **Single:** Specifies single precision (32 bits).
    - **Double:** Specifies double precision (64 bits).
    - **Custom:** Activates the field below so you can specify the Exponent width and the Fraction width.
    - **Exponent width:** Specify the exponent width.
    - **Fraction width:** Specify the fraction width.
- **Quantization:**

  Quantization errors occur when the number of fractional bits is insufficient to represent the fractional portion of a value. The options are to **Truncate** (for example, to discard bits to the right of the least significant representable bit), or to **Round(unbiased: +/- inf)** or **Round (unbiased: even values).**

  Round(unbiased: +/- inf) also known as "Symmetric Round (towards +/- inf)" or "Symmetric Round (away from zero)". This is similar to the MATLAB `round()` function. This method rounds the value to the nearest desired bit away from zero and when there is a value at the midpoint between two possible rounded values, the one with the larger magnitude is selected. For example, to round 01.0110 to a Fix_4_2, this yields 01.10, since 01.0110 is exactly between 01.01 and 01.10 and the latter is further from zero.
- **Overflow:**

  Overflow errors occur when a value lies outside the representable range. For overflow the options are to **Saturate** to the largest positive/smallest negative value, to **Wrap** (for example, to discard bits to the left of the most significant representable bit), or to **Flag as error** (an overflow as a Simulink error) during simulation. **Flag as error** is a simulation only feature. The hardware generated is the same as when **Wrap** is selected.
- **Implementation Tab:**

  Parameters specific to the Implementation Tab are as follows:

    - **Interface Options:**
        - **Interface:**
            - **None:** Implies that during HDL Netlist generation, this Vector Real Gateway In is translated as an Input Port at the top level.
            - **AXI4-Lite:** Implies that during HDL Netlist generation, an AXI4-Lite interface will be created, and this Vector Real Gateway In is mapped to one of the registers within the AXI4-Lite interface.
        - **Auto assign address offset:**

If the Vector Real Gateway In is configured to be an AXI4-Lite interface, this option allows an address offset to be automatically assigned to the register within the AXI4-Lite interface that the Vector Real Gateway In is mapped to.

- **Address offset:**

  If Auto assign address offset is not checked, then this entry box allows you to explicitly specify an address offset to use. Must be a multiple of 4.

- **Interface Name:**

  If the Vector Real Gateway In is configured to be an AX4-Lite interface, assigns a unique name to this interface. This name can be used to differentiate between multiple AXI4-Lite interfaces in the design. When using the IP catalog flow, you can expect to see an interface in the IP that Model Composer creates with the name `<design_name>_<interface_name>_ s_axi`.

  IMPORTANT! *The* ***Interface Name*** *must be composed of alphanumeric characters (lowercase alphabetic) or an underscore (_) only, and must begin with a lowercase alphabetic character. axi4_lite1 is acceptable, 1AXI4-Lite is not.*

- **Description:**

  Additional designer comments about this Vector Real Gateway In that is captured in the interface documentation.

- **Default value:**

- **Constraints:**

  - **IOB Timing Constraint:**

    In hardware, a Vector Real Gateway In is realized as a set of input/output buffers (IOBs). There are three ways to constrain the timing on IOBs. They are None, Data Rate, and Data Rate, Set 'FAST' Attribute.

    - If **None** is selected, no timing constraints for the IOBs are put in the user constraint file produced by Model Composer. This means the paths from the IOBs to synchronous elements are not constrained.
    - If **Data Rate** is selected, the IOBs are constrained at the data rate at which the IOBs operate. The rate is determined by System Clock Period provided on the System Generator token and the sample rate of the Gateway relative to the other sample periods in the design.
    - If **Data Rate, Set 'FAST' Attribute** is selected, the constraints described above are produced. In addition, a FAST slew rate attribute is generated for each IOB. This reduces delay but increases noise and power consumption.

- **Specify IOB location constraints:** Checking this option allows IOB location constraints and I/O standards to be specified.
- **IOB pad locations, e.g. {'MSB', ..., 'LSB'}:** IOB pin locations can be specified as a cell array of strings in this edit box. The locations are package-specific.IO Standards, e.g. {'MSB', ..., 'LSB'}
- **IO Standards, e.g. {'MSB', ..., 'LSB'}:** I/O standards can be specified as a cell array of strings in this edit box. The locations are package-specific.

Vector Real Gateway Out

The Vector Real Gateway Out block converts Xilinx® fixed-point or floating-point type vector inputs into vector outputs of type Simulink® integer, single, double, or fixed-point.

Hardware notes: In hardware these blocks become top level output ports or are discarded depending on how they are configured.

Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This blocks enable 1-D vector support for the primary block operation.

Vector Real Gateway Out blocks are the outputs from the HDL portion of your Simulink design. This block converts the Model Composer fixed-point or floating-point data type into a SimulinkSimulink integer, single, double, or fixed-point data type.

According to its configuration, the Vector Real Gateway Out block can either define an output port for the top level of the HDL design generated by Model Composer, or be used simply as a test point that is trimmed from the hardware representation

Gateway Blocks

As listed below, the Vector Real Gateway Out block is used to provide the following functions:

- Convert data from a Model Composer fixed-point or floating-point data type into a Simulink integer, single, double, or fixed-point data type.
- Define I/O ports for the top level of the HDL design generated by Model Composer. A Vector Real Gateway Out block defines a top-level output port.

- Define test bench result vectors when the Model Composer **Create Testbench** box is checked. In this case, during HDL code generation, the outputs from the block that occur during Simulink simulation are logged as logic vectors in a data file. For each top level port, an HDL component is inserted in the top-level test bench that checks this vector against expected results during HDL simulation.
- Name the corresponding output port on the top-level HDL entity.

Block Parameters

- **Basic Tab :**

  Parameters specific to the Basic Tab are as follows.

  - **Propagate data type to output:** This option is useful when you instantiate a Model Composer design as a sub-system into a Simulink design. Instead of using a Simulink double as the output data type by default, the Model Composer data type is propagated to an appropriate Simulink data type according to the following table:

    Table 61: **Model Composer Data Type Propagation**

Model Composer Data Type	Simulink Data Type
XFloat_8_24	single
XFloat_11_53	double
Custom floating-point precision data type exponent width and fraction width less than those for single precision	single
Custom floating-point precision data type with exponent width or fraction width greater than that for single precision	double
XFix_<width>_<binpt>	sfix<width>_EN<binpt>
UFix_<width>_<binpt>	ufix<width>_EN<binpt>
XFix_<width>_0 where width is 8, 16 or 32	int<width> where width is 8, 16 or 32
UFix_<width>_0 where width is 8, 16 or 32	uint<width> where width is 8, 16 or 32
XFix_<width>_0 where width is other than 8, 16 or 32	sfix<width>
UFix_<width>_0 where width is other than 8, 16 or 32	ufix<width>

  - **Translate into Output Port:** Having this box unchecked prevents the gateway from becoming an actual output port when translated into hardware. This checkbox is on by default, enabling the output port. When this option is not selected, the Vector Real Gateway Out block is used only during debugging, where its purpose is to communicate with Simulink Sink blocks for probing portions of the design. In this case, the Vector Real Gateway Out block turns gray in color, indicating that the gateway will not be translated into an output port.

- **Implementation Tab :** Parameters specific to the Implementation Tab are as follows.

  - **Interface Options:**

        - **None:** During HDL Netlist generation, this Vector Real Gateway Out will be translated as an Output Port at the top level.
        - **AXI4-Lite:** During HDL Netlist Generation, an AXI4-Lite interface will be created, and the Vector Real Gateway Out will be mapped to one of the registers within the AXI4-Lite interface.
        - **Interrupt:** During an IP catalog Generation, this Vector Real Gateway Out is tagged as an Interrupt output port when the Model Composer design is packaged into an IP module that can be included in the Vivado® IP catalog.
    - **Auto assign address offset:**

      If a Vector Real Gateway Out is configured to be an AXI4-Lite interface, this option allows an address offset to be automatically assigned to the register within the AXI4-Lite interface that the Vector Real Gateway Out is mapped to.
    - **Address offset:**

      If Auto assign address offset is not checked, then this entry box allows you to explicitly specify a address offset to use. Must be a multiple of 4.
    - **Interface Name:**

      If the Vector Real Gateway Out is configured to be an AX4-Lite interface, assigns a unique name to this interface. This name can be used to differentiate between multiple AXI4-Lite interfaces in the design. When using the IP catalog flow, you can expect to see an interface in the IP that Model Composer creates with the name `<design_name>_<interface_name>_ s_axi`.

      IMPORTANT! *The **Interface Name** must be composed of alphanumeric characters (lowercase alphabetic) or an underscore (_) only, and must begin with a lowercase alphabetic character.* `axi4_lite1` *is acceptable, 1AXI4-Lite is not.*
    - **Description:**

      Additional designer comments about this Vector Real Gateway Out that is captured in the interface documentation.
- **Constraints:**
    - **IOB Timing Constraint:**

      In hardware, a Vector Real Gateway Out is realized as a set of input/output buffers (IOBs). There are three ways to constrain the timing on IOBs. They are None, Data Rate, and Data Rate, Set 'FAST' Attribute.
    - **None:** No timing constraints for the IOBs are put in the user constraint file produced by Model Composer. This means the paths from the IOBs to synchronous elements are not constrained.

- **Data Rate:**

  The IOBs are constrained at the data rate that the IOBs operate. The rate is determined by System Clock Period provided on the System Generator token and the sample rate of the Gateway relative to the other sample periods in the design. For example, the following OFFSET = OUT constraints are generated for a Vector Real Gateway Out named 'Dout' that is running at the system period of 10 ns:

  ```
  # Offset out constraints
  NET "Dout(0)" OFFSET = OUT : 10.0 : AFTER "clk";
  NET "Dout(1)" OFFSET = OUT : 10.0 : AFTER "clk";
  NET "Dout(2)" OFFSET = OUT : 10.0 : AFTER "clk";
  ```

  - **Specify IOB Location Constraints:** Checking this option allows IOB location constraints to be specified.
  - **IOB Pad Locations, e.g. {'MSB', ..., 'LSB'}:** IOB pin locations can be specified as a cell array of strings in this edit box. The locations are package-specific.

- **Data Rate, Set 'FAST' Attribute:**

  The OFFSET = OUT constraints described above are produced. In addition, a FAST slew rate attribute is generated for each IOB. This reduces delay but increases noise and power consumption. For the previous example, the following additional attributes are added to the constraints file

  ```
  NET "Dout(0)" FAST;
  NET "Dout(1)" FAST;
  NET "Dout(2)" FAST;
  ```

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Vector Real Mult

The Vector Real Multiplier block supports the multiplication feature for vector type inputs.

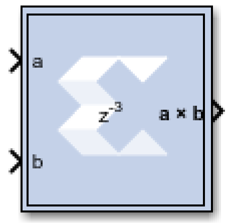
Hardware notes: To check for the optimum internal pipeline stages of the dedicated multiplier select 'Test for optimum pipelining'.

Optimization Goal: For implementation into device fabric (LUTs), the Speed or Area optimization takes effect only if it is supported by IP for the particular device family. Otherwise, the results will be identical regardless of the selection.

Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This block enables 1-D vector support for the primary block operation.

The Vector Real Mult block implements a multiplier. It computes the product of the data on its two input ports, producing the result on its output port.



Block Parameters

Open the Block Parameters dialog box by double-clicking the icon in your Simulink® model.

- **Basic tab:**

  Parameters specific to the Basic tab are as follows.

  - **Precision:**

    This parameter allows you to specify the output precision for fixed-point arithmetic. Floating-point output always has **Full** precision.

    - **Full:** The block uses sufficient precision to represent the result without error.
    - **User Defined:** If you do not need full precision, this option allows you to specify a reduced number of total bits and/or fractional bits.

  - **User-Defined Precision:**

    - **Fixed-point Precision:**

      - **Signed (2's comp):** The output is a Signed (2's complement) number.
      - **Unsigned:** The output is an Unsigned number.
      - **Number of bits:** Specifies the bit location of the binary point of the output number, where bit zero is the least significant bit.
      - **Binary point:** Position of the binary point in the fixed-point output.

  - **Quantization:** Refer to the Overflow and Quantization section in the Common Options in Block Parameter Dialog Boxes topic.

  - **Overflow:**

    Refer to the Overflow and Quantization section inthe Common Options in Block Parameter Dialog Boxes topic.

- **Optional Port:** Provide enable port.
- **Latency:** This defines the number of sample periods by which the block's output is delayed.
- **Saturation and Rounding of User Data Types in a Multiplier:** When saturation or rounding is selected on the user data type of a multiplier, latency is also distributed so as to pipeline the saturation/rounding logic first, and then additional registers are added to the core. For example, if a latency of three is selected, and rounding/saturation is selected, then the first register is placed after the rounding or saturation logic, and two registers are placed to pipeline the core. Registers are added to the core until optimum pipelining is reached and then further registers are placed after the rounding/saturation logic. However, if the data type you select does not require additional saturation/rounding logic, then all the registers are used to pipeline the core.
- **Implementation tab :**

  Parameters specific to the Implementation tab are as follows:

  - **Use behavioral HDL (otherwise use core):**

    The block is implemented using behavioral HDL. This gives the downstream logic synthesis tool maximum freedom to optimize for performance or area.

    Note: For Floating-point operations, the block always uses the Floating-point Operator core.

  - **Core Parameters:**
    - **Optimize for Speed|Area:** Directs the block to be optimized for either Speed or Area.
      - **Use embedded multipliers:** This field specifies that if possible, use the XtremeDSP slice (DSP48 type embedded multiplier) in the target device.
      - **Test for optimum pipelining:** Checks if the Latency provided is at least equal to the optimum pipeline length. Latency values that pass this test imply that the core produced is optimized for speed.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE™ Documentation

LogiCORE IP Multiplier v12.0

LogiCORE IP Floating-Point Operator v7.1

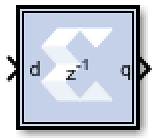
Vector Register

The Vector Register block supports vector type inputs.

Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This block enables 1-D vector support for the primary block operation.

The Vector Register block models a D flip-flop-based register, having latency of one sample period.



Block Interface

The block has one input port for the data and an optional input reset port. The initial output value is specified by you in the block parameters dialog box (below). Data presented at the input will appear at the output after one sample period. Upon reset, the register assumes the initial value specified in the parameters dialog box.

The Vector Register block differs from the Xilinx Delay block by providing an optional reset port and a user specifiable initial value.

Block Parameters

Open the block parameters dialog box by double-clicking the icon in your Simulink® model.

- **Basic Tab:**

  Parameters specific to the Basic tab are as follows.

  - **Initial value:** Specifies the initial value in the register.
  - **Optional Ports:**
    - Provide synchronous reset port.
    - Provide enable port.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Xilinx LogiCORE

The Vector Register block is implemented as a synthesizable VHDL module. It does not use a Xilinx LogiCORE™.

Vector Reinterpret

The Vector Reinterpret block changes the vector input signal type without altering the binary representation. You can change the signal between signed and unsigned, and relocate the binary point.

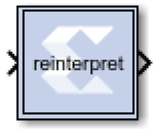
Hardware notes: In hardware this block costs nothing.

Example: The input is 6 bits wide, signed with 2 fractional bits, and the output is forced to unsigned with 0 fractional bits. Then an input of -2.0 (1110.00 in binary 2's complement) becomes an output of 56 (111000 in binary).

Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This block enables 1-D vector data support for the primary block operation.

The Vector Reinterpret block forces its output to a new type without any regard for retaining the numerical value represented by the input.



The binary representation is passed through unchanged, so in hardware this block consumes no resources. The number of bits in the output will always be the same as the number of bits in the input.

The block allows for unsigned data to be reinterpreted as signed data, or, conversely, for signed data to be reinterpreted as unsigned. It also allows for the reinterpretation of the data's scaling, through the repositioning of the binary point within the data. The Xilinx Scale block provides an analogous capability.

An example of this block's use is as follows: if the input type is 6 bits wide and signed, with 2 fractional bits, and the output type is forced to be unsigned with 0 fractional bits, then an input of -2.0 (1110.00 in binary, two's complement) would be translated into an output of 56 (111000 in binary).

This block can be particularly useful in applications that combine it with the Xilinx Slice block or the Xilinx Concat block. To illustrate the block's use, consider the following scenario:

Given two signals, one carrying signed data, and the other carrying two unsigned bits (a `UFix_2_0`), we want to design a system that concatenates the two bits from the second signal onto the tail (least significant bits) of the signed signal.

We can do so using two Vector Reinterpret blocks and one Vector Concat block. The first Vector Reinterpret block is used to force the signed input signal to be treated as an unsigned value with its binary point at zero. The result is then fed through the Vector Concat block along with the other signal's `UFix_2_0`. The Concat operation is then followed by a second Vector Reinterpret that forces the output of the Vector Concat block back into a signed interpretation with the binary point appropriately repositioned.

Though three blocks are required in this construction, the hardware implementation is realized as simply a bus concatenation, which has no cost in hardware.

Block Parameters

Parameters specific to the block are as follows.

- **Force Arithmetic Type:** When checked, the Output Arithmetic Type parameter can be set and the output type is forced to the arithmetic type chosen according to the setting of the Output Arithmetic Type parameter. When unchecked, the arithmetic type of the output is unchanged from the arithmetic type of the input.
- **Output Arithmetic Type:** The arithmetic type (unsigned or signed, 2's complement, Floating-point) to which the output is to be forced.
- **Force Binary Point:** When checked, the Output Binary Point parameter can be set and the binary point position of the output is forced to the position supplied in the Output Binary Point parameter. When unchecked, the arithmetic type of the output is unchanged from the arithmetic type of the input.
- **Output Binary Point:** The position to which the output's binary point is to be forced. The supplied value must be an integer between zero and the number of bits in the input (inclusive).

LogiCORE Documentation

LogiCORE IP Floating-Point Operator v7.1

Vector Relational

The Vector Relational block implements comparator for vector inputs.

Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This block enables 1-D vector data support for the primary block operation.

Block Parameters

The block parameters dialog box can be invoked by double-clicking the icon in your Simulink® model.

The only parameter specific to the Vector Relational block is:

- **Comparison**: specifies the comparison operation computed by the block.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP Floating-Point Operator v7.1

Vector Slice

The Vector Slice block supports vector type inputs.

Extracts a given range of bits from each sample of input vector and presents it at the output. The output type is ordinarily unsigned with binary point at zero, but can be Boolean when the slice is one bit wide.

Hardware notes: In hardware this block costs nothing.

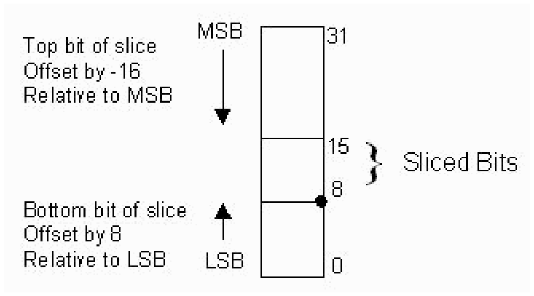
Description

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This block enables 1-D vector support for the primary block operation.

The Vector Slice block allows you to slice off a sequence of bits from your input data and create a new data value. This value is presented as the output from the block. The output data type is unsigned with its binary point at zero.

The block provides several mechanisms by which the sequence of bits can be specified. If the input type is known at the time of parameterization, the various mechanisms do not offer any gain in functionality. If, however, a Vector Slice block is used in a design where the input data width, or binary point position are subject to change, the variety of mechanisms becomes useful. The block can be configured, for example, always to extract only the top bit of the input, or only the integral bits, or only the first three fractional bits. The following diagram illustrates how to extract all but the top 16 and bottom 8 bits of the input.

Figure 385: **Extracting Top 16 and Bottom 8 Bits**



Block Parameters

Open the Block Parameters dialog box by double-clicking the icon in your Simulink® model.

Parameters specific to the block are as follows.

- **Width of slice (Number of bits):** Specifies the number of bits to extract.
- **Boolean output:** Tells whether single bit slices should be type Boolean.
- **Specify range as:** (Two bit locations | Upper bit location + width |Lower bit location + width). Allows you to specify either the bit locations of both end-points of the slice, or one end-point along with number of bits to be taken in the slice.
- **Offset of top bit:** Specifies the offset for the ending bit position from the LSB, MSBm. or binary point.
- **Offset of bottom bit:** Specifies the offset for the ending bit position from the LSB, MSBm, or binary point.
- **Relative to:** Specifies the bit slice position relative to the MSB, LSB, or binary point of the top or the bottom of the slice.

Other parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Vector Up Sample

The Vector Up Sample block up samples input vector data. Inserted values can be zeros or copies of the most recent input sample.

Hardware notes: No hardware is needed if inserted values are copies of the input sample. Otherwise, a mux, and single bit flip-flop are used.

Description

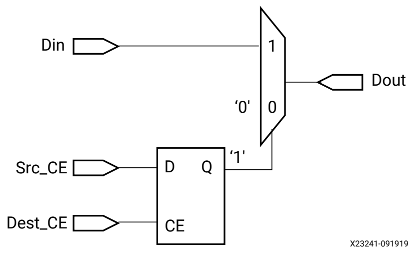
Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This blocks enable 1-D vector support for the primary block operation.

The Vector Up Sample block increases the sample rate at the point where the block is placed in your design. The output sample period is l/n, where l is the input sample period, and n is the sampling rate.

The input signal is up sampled so that within an input sample frame, an input sample is either presented at the output n times if samples are copied, or presented once with (n-1) zeroes interspersed if zero padding is used.

In hardware, the Vector Up Sample block has two possible implementations. If the Copy Samples option is selected on the block parameters dialog box, the Din port is connected directly to Dout and no hardware is expended. Alternatively, if zero padding is selected, a mux is used to switch between the input sample, and inserted zeros. The corresponding circuit for the zero padding Vector Up Sample block is shown below.

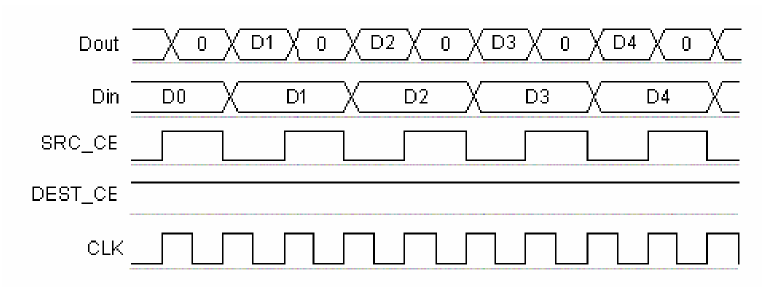
Figure 386: **Zero Padding Up Sample Circuit**



Block Interface

The Vector Up Sample block receives two clock enable signals, Src_CE, and Dest_CE. Src_CE is the clock enable signal corresponding to the input data stream rate. Dest_CE is the faster clock enable, corresponding to the output data stream rate. Notice that the circuit uses a single flip-flop in addition to the mux. The flip-flop is used to adjust the timing of Src_CE, so that the mux switches to the data input sample at the start of the input sample period, and switches to the constant zero after the first input sample. It is important to notice that the circuit has a combinational path from Din to Dout. As a result, a Vector Up Sample block configured to zero pad should be followed by a register whenever possible.

Figure 387: **Up Sample Output**



Block Parameters

Open the Block Parameters dialog box by double-clicking the icon in your Simulink® model.

- **Basic tab:** Parameters specific to the Basic tab are as follows.
    - **Sampling rate (number of output samples per input sample):** Must be an integer with a value of 2 or greater. This is the ratio of the output sample period to the input, and is essentially a sample rate multiplier. For example, a ratio of 2 indicates a doubling of the input sample rate. If a non-integer ratio is desired, the Vector Up Sample block can be used in combination with the Vector Down Sample block.
    - **Copy samples (otherwise zeros are inserted):** Allows you to choose what to do with the additional samples produced by the increased clock rate. By selecting Copy Samples, the same sample is duplicated (copied) during the extra sample times. If this checkbox is not selected, the additional samples are zero.
    - **Provide enable port:** When checked, this option adds an en (enable) input port, if the Latency is specified as a positive integer greater than zero.

- **Latency:** This defines the number of sample periods by which the block's output is delayed. One sample period can correspond to multiple clock cycles in the corresponding FPGA implementation (for example, when the hardware is over-clocked with respect to the Simulink model). The user defined sample latency is handled in the Vector Up Sample block by placing shift registers that are clock enabled at the input sample rate, on the input of the block. The behavior of a Vector Up Sample block with non-zero latency is similar to putting a delay block, with equivalent latency, at the input of an Vector Up Sample block with zero latency.

Parameters used by this block are explained in the topic Common Options in Block Parameter Dialog Boxes.

Vector2Scalar

The Vector2Scalar block converts vector type input to scalar type output.

Description

The Vector2Scalar block does the bit level concatenation of all the elements of the input vector to produce a scalar output.

I
Vector2Scalar
O

For example, if the input vector is [0 1 2 7] of type `Ufix_3_0`, and the SSR parameter is 4, it produces Scalar output 3720 of type `Ufix_12_0`, whose binary value is 111 010 001 000. This value represents input vector when we split it into four groups each of 3 bits.

Data Type Support

- The inputs must be Boolean or unsigned fixed-point signal.
- All inputs must have binary set to 0.

Parameters

Super Sample Rate (SSR): This configurable GUI parameter is primarily used to control processing of multiple data samples on every sample period. This block enables 1-D vector support for the primary block operation.

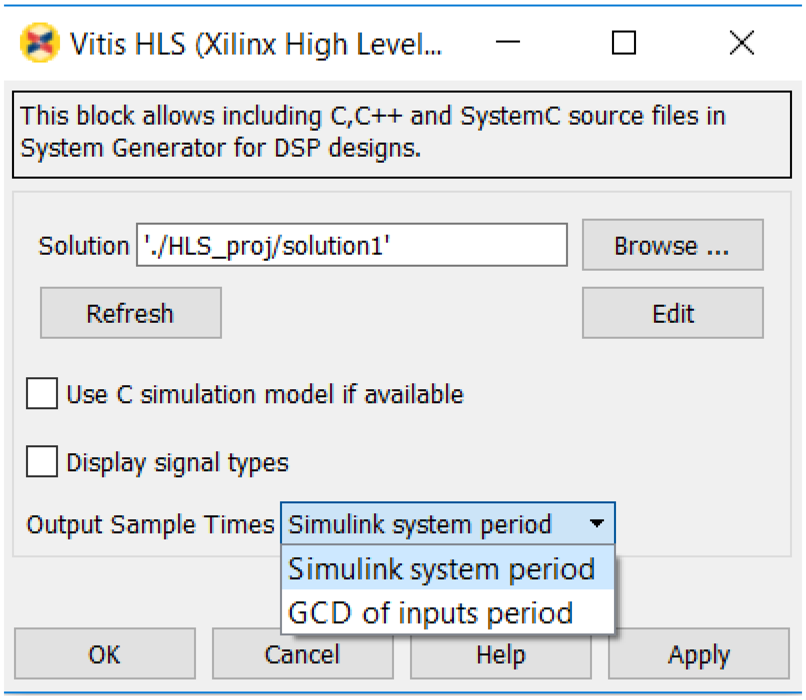
Vitis HLS

The Xilinx Vitis™ HLS block allows the functionality of a Vitis HLS design to be included in a Model Composer design. The Vitis HLS design can include C, C++ and System C design sources.

There are two steps to the method of including a Vitis HLS design into Model Composer. The first step is to use the Vitis HLS RTL Packaging feature to package the design files into a Solution directory. (Refer to Vitis HLS documentation for more information regarding RTL Packaging.) The second step is to place the Vitis HLS block in your Model Composer design and specify the Vitis HDL Solution directory as the target.

Block Parameters Dialog Box

Figure 388: **Block Parameters Dialog Box**



- **Solution:** The path to the Solution space directory containing RTL packaged for Model Composer. This path is usually the path to a directory contained in a Vivado® HLS project. The path must be included in single quotes and must evaluate to a string.
- **Browse:** A standard directory browse button.
- **Refresh:** Updates the block ports to the latest package contained in the solution space.
- **Edit:** Opens the Vitis HLS project associated with solution space.
- **Use C simulation model if available:** Use the C simulation model if it is available in the Vitis HLS package. As shown below, the simulation model being used is shown on the Vitis HLS block. In this case, an RTL-model is used because a C simulation model is not available.

- **Display signal types:** Signal types to be used to drive input ports and emanating from output ports are displayed on the block icon when checked.
- **Output Sample Times:** Select either the **Simulink system period** or the **GCD of the inputs period**.

Data Type Translation

Data Type Translation	
C/C++ Data Type	**Model Composer Data Type**
float	XFloat_32_23
double	XFloat_64_52
bool	UFix_1_0
(unsigned) char	(U)Fix_8_0
(unsigned) short	(U)Fix_16_0
(unsigned) int	(U)Fix_32_0
(unsigned) long	(U)FIX_<PlatformDependent>_0
(unsigned) long long	(U)Fix_64_0
ap_(u)fix<N,M>	(U)Fix_<N>_<N-M>
ap_(u)int<N>	(U)Fix_N_0

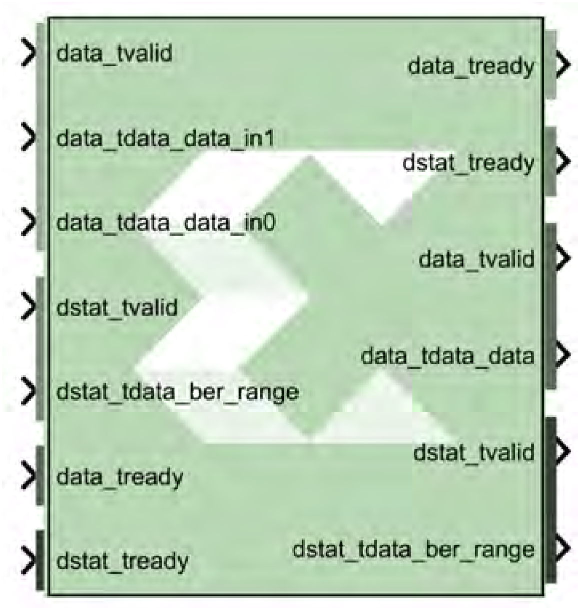
Known Issues

- It is not possible to include a purely combinational design from Vitis HLS. The design must synthesize into an RTL design that contains a Clock and a Clock Enable input.
- The top-level module cannot contain C/C++ templates.
- Composite ports will be represented as UFix_<N>_0 only where N is the width of the port.
- The current C simulation model only supports fixed latency and interval designs. The latency and interval numbers are obtained from the synthesis engine.
- The current C simulation model supports the default block-level communication protocol (ap_hs).
- The current C simulation model does not support the ‘ap_memory’ and ‘ap_bus’ interfaces.
- Vitis HLS block does not support combinational designs due to performance considerations. In the current implementation, Model Composer updates each HLS input port multiple times every clock cycle. So it is very costly to evaluate the DUT whenever inputs changes.
- The output values match RTL simulation results only when corresponding control signals indicate data are valid. So test bench and downstream blocks should read/observe data based on the communication protocol and control signals.
- Because the Vitis HLS block has to use the GCC shipped in the Vivado Design Suite to compile dll on Win-64 platform, users cannot use arbitrary bitwidth integers in C designs on win-64 systems.

Viterbi Decoder 9.1

***Note*:** This block goes into the FPGA fabric and is a Licensed Core. Please visit the Xilinx web site to purchase the appropriate core license.

Data encoded with a convolution encoder can be decoded using the Xilinx Viterbi decoder block. This block adheres to the AXI4-Stream standard.



There are two steps to the decode process. The first weighs the cost of incoming data against all possible data input combinations; either a Hamming or Euclidean metric can be used to determine the cost. The second step traces back through the trellis and determines the optimal path. The length of the trace through the trellis can be controlled by the traceback length parameter.

The decoder achieves minimal error rates when using optimal convolution codes; the table below shows various optimal codes. For correct operation, convolution codes used for encoding must match with that for decoding.

Table 62: **Convolution Codes**

Constraint Length	Optimal Convolution Codes for 1/2 rate (octal)	Optimal Convolution Codes for 1/3 Rate (octal)
3	[7 5]	[7 7 5]
4	[17 13]	[17 13 15]
5	[37 33]	[37 33 25]
6	57 65]	[57 65 71]
7	[117 127]	[117 127 155]
8	[357 233]	[357 233 251]

Table 62: **Convolution Codes** *(cont'd)*

Constraint Length	Optimal Convolution Codes for 1/2 rate (octal)	Optimal Convolution Codes for 1/3 Rate (octal)
9	[755 633]	[755 633 447]

Block Interface

The Xilinx Viterbi Decoder 9.1 block is AXI4 compliant. The following describes the standard AXI channels and pins on the interface.

- **S_AXIS_DATA Channel:**
  - **s_axis_data_tvalid:** TVALID for S_AXIS_DATA channel. Input pin, always available. This port indicates the values presents on the input data ports are valid.
  - **s_axis_data_tready:** TREADY for S_AXIS_DATA. Output pin, always available. This port indicates that the core is ready to accept data.
  - **s_axis_data_tdata:** Input TDATA. Different input data ports are available depending on the Viterbi Type selected on Page1 tab of block-GUI.

  When Trellis Mode is selected, 5 input data pins become available – these are **s_axis_data_tdata_tcm00**, **s_axis_data_tdata_tcm01**, **s_axis_data_tdata_tcm10**, **s_axis_data_tdata_tcm11** and **s_axis_data_tdata_sector**.

  The width of the Trellis mode inputs (s_axis_data_tdata_tcm**) can range from 4 to 6 corresponding to a data width (Soft_Width value on Page2 tab) of 3 to 5. s_axis_data_tdata_sector is always 4-bit wide. The decoder always functions as a rate 1/2 decoder when Trellis mode is selected.

  For any other Viterbi Type (Standard/Multi-Channel/Dual Decoder), the Decoder supports rates from 1/2 to 1/7. Therefore, the block can have 2 to 7 input data ports labeled **s_axis_data_tdata_data_in0 s_axis_data_tdata_data_in6**. Hard Coding requires each tdata_data_in<n> port to be 1 bit wide. Soft Coding allows these widths to be between 3 to 5 bits (inclusive).

  - **s_axis_data_tuser:** TUSER for S_AXIS_DATA. These ports are only present if External Puncturing is selected or it is a Dual Decoder or Block Valid signal is used with the core.
    - **s_axis_data_tuser_erase:** Port becomes available, when External Puncturing is selected (on Page2 tab). This input bus is used to indicate the presence of a null-symbol on the corresponding data_in buses. For e.g. tuser_erase(0) corresponds to data_in0, tuser_erase(1) corresponds to data_in1 etc. If an erase bit is high, the data on the corresponding data_in bus is treated as a null-symbol internally to the decoder. The width of the erase bus is equal to the output rate of the decoder with a maximum value of 7.

    - **s_axis_data_tuser_sel:** Port becomes available when Dual Decoder is selected. This is used to select the correct set of convolution codes for the decoding of the input data symbols in the dual decoder case. When SEL is low, the input data is decoded using the first set of convolution codes. When it is high, the second set of convolution codes is applied.
    - **s_axis_data_tuser_block_in:** Port becomes available when Block Valid option is selected on Page 5 tab.
- **M_AXIS_DATA Channel:**
  - **m_axis_data_tvalid:** TVALID for M_AXIS_DATA channel. Output pin, always available. It indicates whether the output data is valid or not.
  - **m_axis_data_tready:** TREADY for M_AXIS_DATA channel. Do not enable or tie high if downstream slave is always able to accept data. It becomes available when TREADY option is selected on Page 5 tab.
  - **m_axis_data_tdata:** Decoded TDATA for output data channel.
    - **m_axis_data_tdata_data:** Port represents the decoded output data and it is always 1 bit wide.
    - **m_axis_data_tdata_sector:** Port becomes available for Trellis Mode decoder. This port is always 4-bit wide. The output SECTOR is a delayed version of the input SECTOR bus. Both buses have a fixed width of 4 bits. The delay equals the delay through the Trellis Mode decoder.
  - **m_axis_data_tuser:** TUSER for M_AXIS_DATA channel. These ports are only present if the block is a Dual Decoder or it has normalization signal present or it has Block Valid option checked.
    - **m_axis_data_tuser_sel:** Port becomes available when the block is configured as a Dual Decoder. This signal is a delayed version of the input s_axis_data_tuser_sel signal. The delay equals to the delay through the Dual Decoder.
    - **m_axis_data_tuser_norm:** Port becomes available when NORM option is checked on Page 5 tab. This port indicates when normalization has occurred within the core. It gives an immediate indication of the rate of errors in the channel.
    - **m_axis_data_tuser_block_out:** Port becomes available when Block Valid option is checked on Page 5 tab. This signal is a delayed version of the input s_axis_data_tuser_block_in signal. The BLOCK_OUT signal shows the decoded data corresponding to the original BLOCK_IN set of data points. The delay equals the delay through the decoder.
- **S_AXIS_DSTAT Channel:**

  ***Note*:** These ports become available when Use BER Symbol Count is selected on Page 5 tab.

  - **s_axis_dstat_tvalid:** TVALID for S_AXIS_DSTAT channel.

- **s_axis_dstat_tready:** TREADY for S_AXIS_DSTAT channel. Indicates that the core is ready to accept data. Always high, except after a reset if there is not a TREADY on the output.
- **s_axis_dstat_tdata_ber_range:** TDATA for S_AXIS_DSTAT channel. This is the number of symbols over which errors are counted in the BER block.

- **M_AXIS_DSTAT Channel:**

  ***Note*:** These ports become available when Use BER Symbol Count is selected on Page 5 tab.

  - **m_axis_dstat_tvalid:** TVALID for M_AXIS_DSTAT channel.
  - **m_axis_dstat_tready:** TREADY for M_AXIS_DSTAT channel. Do not enable or tie high if downstream slave is always able to accept data. It becomes available when TREADY option is selected on Page 5 tab.
  - **m_axis_dstat_tdata_ber:** TDATA for M_AXIS_DSTAT channel. The Bit Error Rate (BER) bus output (fixed width 16) gives a measurement of the channel bit error rate by counting the difference between the re-encoded DATA_OUT and the delayed DATA_IN to the decoder.

- **Other Optional Pins:**

  - **aresetn:** The synchronous reset (aresetn) input can be used to re-initialize the core at any time, regardless of the state of aclken signal. aresetn needs to be asserted low for at least two clock cycles to initialize the circuit. This pin becomes available if ARESETN option is selected on the Page 5 tab. It must be of type Bool. If this pin is not selected, Model Composer ties this pin to inactive (high) on the core.
  - **aclken:** Carries the clock enable signal for the decoder. The signal driving aclken must be Bool. This pin becomes available if ACLKEN option is selected on Page 5 tab.

Block Parameters

- **Page1 tab:**

  Parameters specific to the Page1 tab are as follows.

  - **Viterbi Type:**
    - **Number of Channels:** Used with the Muli-Channel selection, the number of channels to be decoded can be any value between 2 and 32.
    - **Standard:** This type is the basic Viterbi Decoder.
    - **Multi-Channel:** This type allows many interlaced channels of data to be decoded using a single Viterbi Decoder.
    - **Trellis Mode:** This type is a trellis mode decoder using the TCM and SECTOR_IN inputs.
    - **Dual Decoder:** When selected, the block behaves as a dual decoder with two sets of convolutional codes. This makes the sel input port available.

- **Decoder Options:**
  - **Use Reduced Latency:** The latency of the block depends on the traceback length and the constraint length. If this reduced latency option is selected, then the latency of the block is approximately halved and the latency is only 2 times the traceback length.
  - **Constraint length:** Equals n+1, where n is the length of the constraint register in the encoder.
  - **Traceback length:** Length of the traceback through the Viterbi trellis. Optimal length is 5 to 7 times the constraint length.

- **Page2 tab:**
  - **Architecture:**
    - **Parallel:** Large but fast Viterbi Decoder.
    - **Serial:** Small but processes the input data in a serial fashion. The number of clock cycles needed to process each set of input symbols depends on the output rate and the soft width of the data.
  - **Best State:**
    - **Use Best State:** Gives improved BER performance for highly punctured data.
    - **Best State Width:** Indicates how many of the least significant bits to ignore when saving the cost used to determine the best state.
  - **Puncturing:**
    - **None:** Input data has not been punctured.
    - **External (Erased Symbols):** When selected an erase port is added to the block. The presence of null-symbols (that is, symbols which have been deleted prior to transmission across the channel) is indicated using the erasure input erase.
  - **Coding:**
    - **Soft Width:** The input width of soft-coded data can be anything in the range 3 to 5. Larger widths require more logic. If the block is implemented in serial mode, larger soft widths also increase the serial processing time.
    - **Soft Coding:** Uses the Euclidean metric to cost the incoming data against the branches of the Viterbi trellis.
    - **Hard Coding:** Uses the Hamming difference between the input data bits and the branches of the Viterbi trellis. Hard coding is only available for the standard parallel block.
  - **Data Format:**

- **Signed Magnitude:**
- **Offset Binary (available for soft coding only):**

See Table 1 in the associated LogiCORE™ Product Specification for the Signed Magnitude and Offset-Binary data format for Soft Width 3.

- **Page3 tab:**
  - **Convolution 0:**
    - **Output Rate 0:** Output Rate 0 can be any value from 2 to 7.
    - **Convolution Code 0 Radix:** The convolutional codes can be input and viewed in binary, octal, or decimal.
    - **Convolution Code Array (0-6):** First array of convolution codes. Output rate is derived from the array length. Between 2 and 7 (inclusive) codes can be entered. When dual decoding is used, a value of 0 (low) on the sel port corresponds to this array.
- **Page4 tab:**

  The options on this tab are activated when you select **Dual Decoder** as the Viterbi Type on the Page1 tab.
  - **Convolution 1:**
    - **Output Rate 1:** Output Rate 1 can be any value from 2 to 7. This is the second output rate used if the decoder is dual. The incoming data is decoded at this rate when the SEL input is high. Output Rate 1 is not used for the non-dual decoder.
    - **Convolution Code 1 Radix:** The convolutional codes can be input and viewed in binary, octal, or decimal.
- **Page5 tab:**
  - **BER Options:**
    - **Use BER Symbol Count:** This bit-error-rate (BER) option monitors the error rate on the transmission channel.
  - **Optional Pins:**
    - **NORM:** Indicates when normalization has taken place internal to the Add Compare Select module.
    - **Block Valid:** Check this box if BLOCK_IN and BLOCK_OUT signals are required. These signals track the movement of a block of data through the decoder. BLOCK_OUT corresponds to BLOCK_IN delayed by the decoder latency.
    - **TREADY:** Selecting this option makes m_axis_data_tready and m_axis_dstat_tready pins available on the block.

- **ACLKEN:** Carries the clock enable signal for the block The signal driving aclken must be Bool.
- **ARESETN:** Adds a aresetn pin to the block. This signal resets the block and must be of type Bool. aresetn must be asserted low for at least 2 clock periods and at least 1 sample period before the decoder can start decoding code symbols.

Common Parameters used by this block, such as **Display shortened port names**, are explained in the topic Common Options in Block Parameter Dialog Boxes.

LogiCORE Documentation

LogiCORE IP Viterbi Decoder v9.1

HLS Blockset

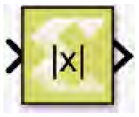
The Vitis Model Composer HLS block library includes the following blocks. The content shown here can also be accessed from the **Help** command for the block within the HLS block library in Simulink.

Abs

Compute element-wise absolute value of input signal

Library

Math Functions/Math Operations



Description

The Abs block computes element-wise absolute value on the input data.

Data Type Support

Data type support is:

- Dimension: Input can be scalar, vector, or matrix.
- Data Types: Input supports signals of integer type, floating point data type (double, single, and half), and signed and unsigned fixed point type.
- Complex Numbers Support: Yes.

Output has the same dimension and data type as the input. Output of the Abs block is always real.

Parameters

The Abs block has no parameters to set.

atan

Compute element-wise arctangent function of an argument.

Library

Math Functions / Math Operations

Description

The atan block returns the output of the atan (`x`) function for each element in array `x`.

Data Type Support

Data types accepted at the inputs of the block are:

- Dimension: Input can be scalar, vector, or matrix.
- Data Types: Input supports signals of integer type, floating point type (double, single and half) and fixed point type.
- Complex Number Support: No

Output has the same dimension and type as the input. However, If the data type of the input is a fixed point type, the data type of the output is fixed point type with integer width fixed as 2. The reason for this is that the output of the atan function is between -π/2 and π/2. Use the atan2 function if you need the output of the function to be between -π and π.

Parameters

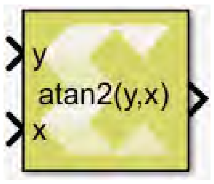
The Atan block has no parameters to set.

atan2

Compute element-wise four-quadrant inverse tangent of input signal.

Library

Math Functions / Math Operations



Description

The atan2 block returns the output of the function atan2(`y,x`).

Data Type Support

Data types accepted at the inputs of the block are:

- Dimension: Inputs can be scalar, vector, or matrix. If one of the inputs is scalar and the other is a vector or matrix then the scalar input is expanded to match the other input dimension, and operation will be performed element wise. If both inputs are non-scalar, then they must match in dimension.
- Data Types: Input supports signals of integer type, floating point type (double, single, and half) and signed and unsigned fixed point type. Both inputs must be of the same data type.
- Complex Number Support: No

Except for fixed-point input data type, output has the same data type as the input. For fixed point input, the output data type will be signed fixed-point with a 3-bit integer width, to be able to represent numbers between $-\pi$ and π.

Parameters

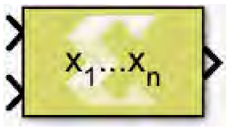
The atan2 block has no parameters to set.

Bit Concat

Perform bitwise concatenation of input values into a single output value

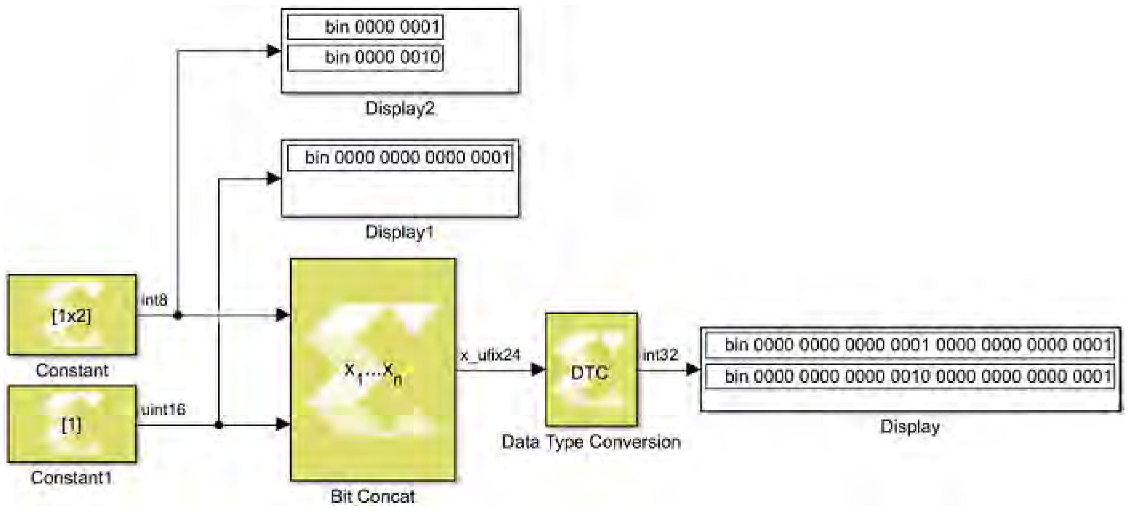
Library

Logic and Bit Operations



Description

Starting from the input with the highest order (the first input at the top in normal block orientation) the bit values of all input ports are concatenated into a single output bit vector. For multidimensional inputs the dimensions of all inputs must match and concatenation proceeds element-wise as it would in the scalar case. A scalar value on one input is automatically expanded to match the dimensions of the other input.



Data Type Support

Data type support for the Bit Concat block is:

- All integer types (including Boolean) and fixed-point types are supported. Floating point types are not supported.
- All inputs must be of real numeric type. Complex types are not supported.
- Scalars, Vectors and 2-D Matrices are supported. Unless an input is a scalar, the dimensions of all inputs must agree.
- The output type is always an unsigned fixed-point type without fractional bits.

Parameters

Number of inputs

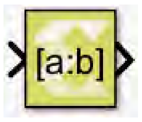
Sets the number of inputs to be concatenated. The minimum number of inputs is 2, the maximum is 128. The sum of all input bit widths shall not exceed 1024 bits.

Bit Slice

Extract a range of bits from a value

Library

Logic and Bit Operations



Description

The Bit Slice block allows the element-wise extraction of a contiguous set of bits from the input values. The extracted bits are returned as unsigned fixed point values of an all-integer range, and you specify the width of the specified extraction range. The block dialog box allows you to specify the range of bits using one of these methods:

- **Bottom bit + width** - You specify the bottom bit and the number of bits to be extracted (**Slice width**).
- **Top bit + width** - You specify the top bit and the number of bits to be extracted (**Slice width**).
- **Top and bottom bit** - You specify the top and bottom bits and the number of bits to be extracted is implied.

The top and bottom bit specifications have multiple ways of specifying the position in relation to either the Least Significant Bit (LSB), the Binary Point of a fixed-point value, or the Most Significant Bit (MSB). In case of integer inputs the Binary Point and Least Significant Bit options are equivalent. Offsets to specify the position relative to these anchors can be positive or negative. However, an error will occur during simulation and/or code generation if the extraction range lies outside of the input type bit range.

Data Type Support

The Bit Slice block accepts any real-valued integer or fixed-point type of any dimension N ≤ 2. Floating point values and complex numeric types are not supported.

The output data type is always a real-valued unsigned fixed-point type with integer-only range. The output data has the same width as the extraction range you specify.

The output dimensions are the same as the input dimensions.

Parameters

Specify range as

The **Specify range as** parameter specifies the extraction range.

- If you select **Bottom bit + width**, parameters in the **Bottom of bit range** section are enabled and parameters in the **Top of bit range** section are disabled.

- If you select **Top bit + width**, parameters in the **Top of bit range** section are enabled and parameters in the **Bottom of bit range** section are disabled.
- If you select **Top and Bottom bit**, parameters in both the **Top of bit range** section and the **Bottom of bit range** section are enabled, and the **Slice width** parameter is disabled.

Following are the settings for the **Specify range as** parameter.

Table 63: **Specify Range As Parameter**

Setting	Description
Bottom bit + width	The **width** specifies the number of bits to extract. The **Bottom bit** of the range specifies the offset at which the range begins (offset of the least significant bit to be extracted).
Top bit + width	The **width** specifies the number of bits to extract. The **Top bit** of the range specifies the offset at which the range begins (offset of the most significant bit to be extracted).
Top and bottom bit	The **Top** bit of the range specifies the offset of the most significant bit to be extracted. The **bottom** bit of the range gives the offset of the least significant bit to be extracted. The width of the extracted range is given implicitly.

Slice width

Specifies the width of the bit range to be extracted. **Slice width** is only enabled if the **Specify range as** parameter is set to **Top bit + width** or **Bottom bit + width**.

Enter a scalar positive integer value for **Slice width**.

Bit position relative to

Defines the basis for offset specifications in both **Top of bit range** and **Bottom of bit range** sections of the block dialog box.

Following are the settings for the **Bit position relative to** parameter.

Table 64: **Bit Position Relative To Parameter**

Setting	Description
Least Significant Bit	Defines the offset parameter as counting from the LSB of the input value, with offset 0 denoting the LSB, offset 1 denoting the bit to the left of the LSB, etc. If the **Least Significant Bit** setting is selected, the **With offset** parameter cannot specify a negative offset.
Binary point	Defines the offset parameter as counting from the binary point of a fixed point value, with offset 0 denoting the least significant integer bit. A negative offset denotes a range starting in the fractional bits with offset -1 being the most significant fractional bit. A positive offset denotes a range starting in the integer portion of the value.
Most Significant Bit	Defines the offset parameter as counting from the MSB of the input value, with offset 0 denoting the MSB, offset -1 denoting the bit to the right of the MSB, etc. If the **Most Significant Bit** stetting is selected, the **With offset** parameter cannot specify a positive offset.

With offset

Specifies the offset to be applied to the basis specified by the corresponding **Bit position relative to** parameter. The **With offset** parameter is available in both **Top of bit range** and **Bottom of bit range** sections of the block dialog box.

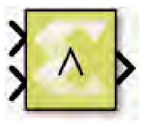
Negative offsets specify bit positions to the right of the anchor (zero offset basis). Positive offsets specify bit positions to the left of the anchor.

Bitwise AND

Perform element and bitwise Boolean AND operation on the inputs

Library

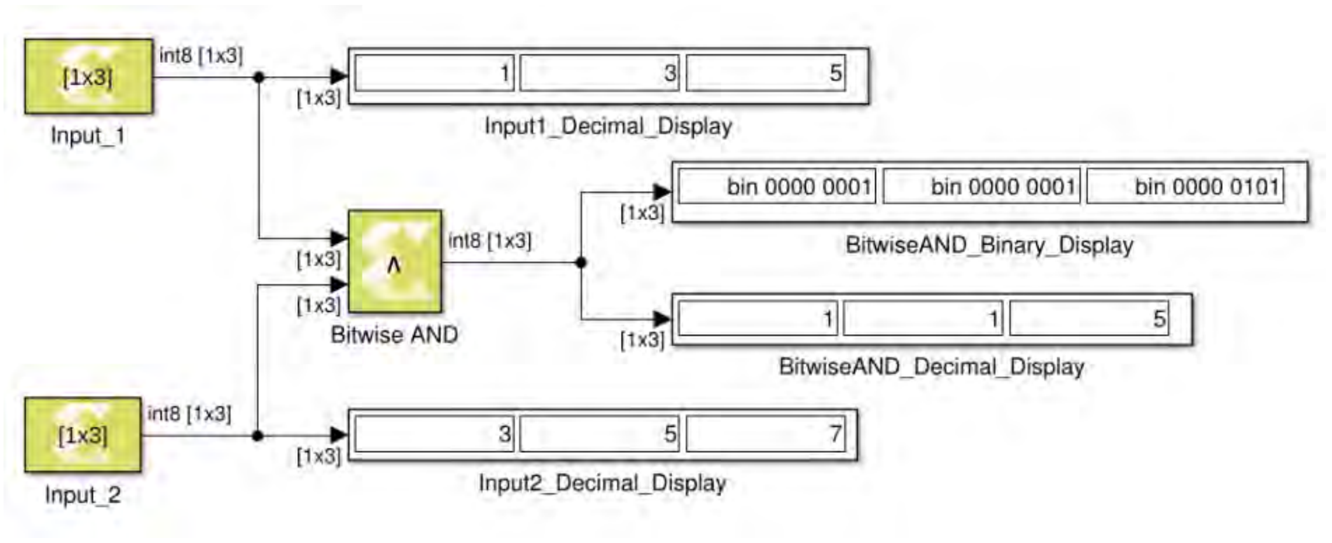
Logic and Bit Operations



Description

The Bitwise AND block has two input signals and one output signal. The block performs element and bit-wise Boolean AND operation on the inputs. The first input corresponds to the top input port and the second input to the bottom input port. Both input ports must have the same data type. The dimension of the output signal matches the dimensions of the input signals. Unless an input is a scalar, the dimensions of all inputs must agree. A scalar value on one input is automatically expanded to match the dimension of the other input.

Figure 389: **Bitwise AND**



Data Type Support

The block accepts integer, fixed-point, and Boolean types of real numeric type. It does not support floating point input types. Complex signals are not supported for this operation.

Parameters

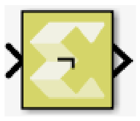
The Bitwise AND block has no parameters to set.

Bitwise NOT

Perform element and bit-wise Boolean NOT operation on the input

Library

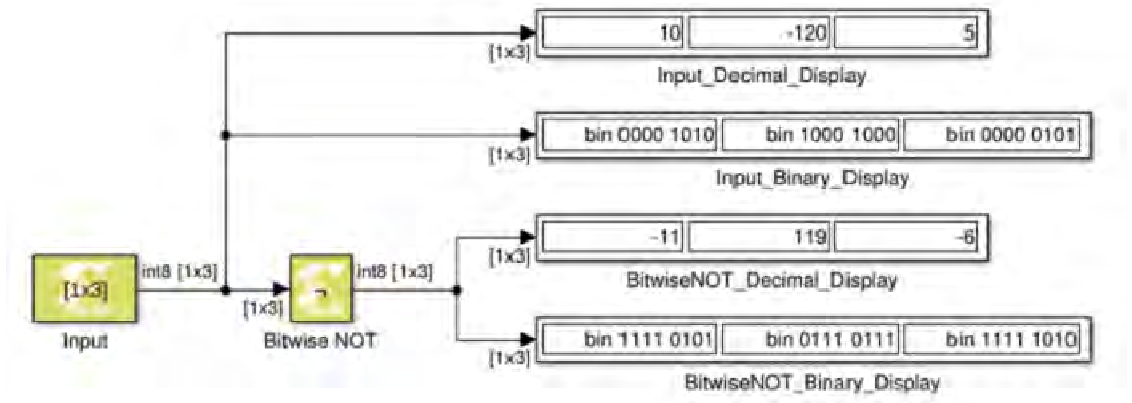
Logic and Bit Operations



Description

The Bitwise NOT block has one input signal and one output signal. It performs element and bit-wise Boolean NOT operation on the input. The dimension of the output signals matches the dimension of the input signals.

Figure 390: **Bitwise NOT**



Data Type Support

The Bitwise NOT block accepts integer, fixed-point, and Boolean types of real numeric type. The block does not support floating point input types. Complex signals are not supported.

Parameters

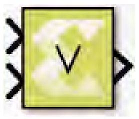
The Bitwise NOT block has no parameters to set.

Bitwise OR

Perform element and bitwise Boolean OR operation on the inputs

Library

Logic and Bit Operations



Description

The Bitwise OR block has two input signals and one output signal. The block performs element and bit-wise boolean OR operation on the inputs. The first input corresponds to the top input port and the second input to the bottom input port. The dimension of the output signal matches the dimensions of the input signals.

Data Type Support

The Bitwise OR block supports native and fixed-point types except the floating point types (double, single, and half). If one of the inputs is scalar, the output dimension is non-scalar. If both inputs are non-scalar, they must have the same dimension. The dimension can be scalar, vector, or matrix.

Parameters

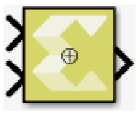
The Bitwise OR block has no parameters to set.

Bitwise XOR

Perform element and bit-wise Boolean XOR operation on the inputs

Library

Logic and Bit Operations



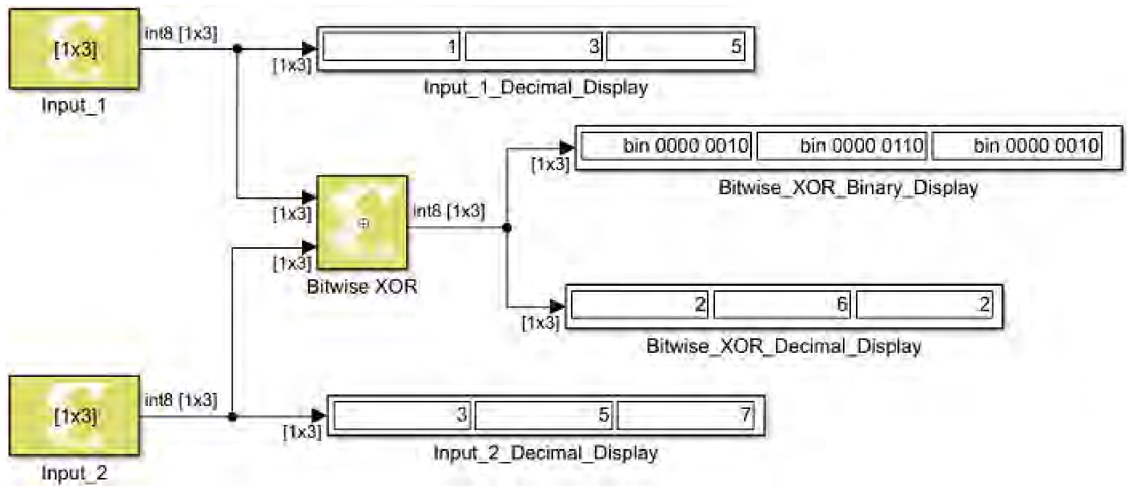
Description

The Bitwise XOR block has two input signals and one output signal. The block performs element and bit-wise Boolean XOR operation on the inputs. The first input corresponds to the upper input port and the second input to the lower input port. The dimension of the output signal matches the dimensions of the input signals.

Data Type Support

If one of the inputs is scalar, the output dimension is non-scalar. If both inputs are non-scalar, they must have the same dimension. The dimension can be scalar, vector, or matrix.

Figure 391: **Bitwise XOR**



Parameters

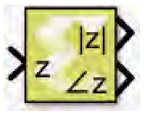
The Bitwise XOR block has no parameters to set.

Complex to Polar

Perform an element-wise conversion of complex input signals into magnitude and radiant phase angle

Library

Math Functions / Math Operations



Description

The Complex to Polar block accepts a complex-valued signal of floating point data types such as double, single and half. It outputs the magnitude and phase angle of the input signal. The outputs are real values of the same data type as the block input. The input can be a scalar, vector or matrix of complex signals, in which case the output signals are also scalar, vector or matrix. The magnitude signal array contains the magnitudes of the corresponding complex input elements. The angle output similarly contains the angles of the input elements in radian.

Data Type Support

Data type support for the input port is:

- Dimension: The input can be scalar, vector or matrix. If the input is not scalar then the outputs have the same dimension as the input.
- Data Types: Input supports complex signal of floating point data type such as double, single, and half.

The outputs are always real-valued signals of floating point data type such as double, single, and half.

Parameters

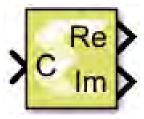
The Complex to Polar block has no parameters to set.

Complex to Real-Imag

Computes the real and imaginary components of the input

Library

Math Functions / Math Operations



Description

The Complex to Real-Imag block accepts a complex or real signal of any valid data type and outputs the real and/or imaginary components of the input signal. The outputs are real-valued signals of the same data type as the block input. The input can be a scalar, vector, or matrix. The outputs have the same dimensions as that of the input.

Data Type Support

Data type support for the input port is:

- Supports real or complex input of any valid data type. The outputs are real-valued signals of the same data type as the block input.
- The input can be a scalar, vector, or matrix. The outputs have the same dimensions as that of the input.

Parameters

- **Output:**

  This parameter specifies the kind of output the block produces.

  Settings for the **Output** parameter are:

Table 65: **Output Parameters**

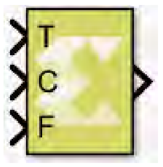
Setting	Description
Real and imag	Outputs real and imaginary parts of the input signal as Re and Im outputs of the block, respectively.
Real	Outputs the real part of the input signal as Re output of the block.
Imag	Outputs imaginary part of the input signal as Im output of the block. If the input is real, the Im output is zero valued.

Conditional

Pass through input T when control input C satisfies a selected criteria; otherwise, pass through input F. The first and third input ports are data ports, and the second input port is the control port.

Library

Signal Routing



Description

The Conditional block passes through input T when control input C satisfies the selected criteria; otherwise, it passes through input F. The first and third input ports are data ports, and the second (or middle) input port is the control port.

Data Type Support

Data type support for the Conditional block is:

- Inputs T and F should either be both complex or real, and can be scalar, vector, or matrix. For non fixed-point data types, inputs T and F do not necessarily have the same data type. But, for fixed point data types, both these inputs should be of fixed-point type. In that case the output data type will be fixed-point, and the number of integer and fractional bits will be set to accommodate both inputs without loss of precision. For example, if one of the inputs is x_sfix16_En10 (1 signed bit, 5 integer bits and 10 fractional bits), and the other is x_sfix16_En5 (1 signed bit, 10 integer bits, and 5 fractional bits), the output will be x_sfix21_En10 (1 signed bit, 10 integer bits, and 10 fractional bits).

- Input C can be a scalar, vector, or matrix if inputs T and F are scalars and Threshold parameter is scalar. In this case, the output dimension matches with the input C dimension. Input C must match the dimension of input T and F if they are non scalars. Input C should be real and can be of any data type..

Parameters

- **Criteria for passing first input:**

  This parameter is used to select the condition under which the block passes the first input (T). If the control input C meets the condition set in the **Criteria for passing first input** parameter, the block passes the first input (T). Otherwise, the block passes the third input (F).

  Settings for the **Criteria for passing first input** parameter are:

  Table 66: **Criteria for Passing First Input Settings**

Setting	Description
C >= Threshold	Select input T if control input C is greater than or equal to the **Threshold** parameter.
C > Threshold	Select input T if control input C is greater than the **Threshold** parameter.
C ~= 0	Select input T if control input C is not equal to 0. Selecting **C ~= 0** disables the **Threshold** parameter.

- **Threshold:**

  This parameter assigns the switch threshold that determines which input the block passes to the output. Threshold parameter is rounded to the same data type as that of the C input.

  - **Settings:**

    Table 67: **Supported Threshold Settings**

Settings	Description
0	default value
`real number`, `vector`, or `matrix`	any real scalar, vector or matrix

  TIP:

  To specify a non-scalar threshold, use brackets. For example, the following entries are valid: [1 3 5].

  For a non-scalar threshold, the inputs must be scalars. In that case, the output dimension is the same as the threshold dimension, and input 2 is compared to each element of the threshold, and depending on the criteria, either T or F is selected to populate the output signal.

  - **Dependencies:** Setting Criteria for passing first input to C ~= 0 disables this parameter.

Conjugate

Apply element-wise complex conjugate operation to the input signal

Library

Math Functions / Math Operations

Description

The Conjugate block applies element-wise complex conjugate operation to the input signal.

The conjugate of a complex number is the number with equal real part, where the imaginary part is equal in magnitude but opposite in sign. The complex conjugate of a+bi is a-bi.

Data Type Support

Data type support for the input is:

- The input signals can be signed integer, fixed-point, or floating-point data type.
- Boolean and unsigned data types are not supported.
- The input signals can be a scalars, vectors, or matrices.
- The input signal supports complex type.

The data type and dimension of the output signal are the same as those of input signal. For complex type input, only the magnitude of the imaginary part changes.

Parameters

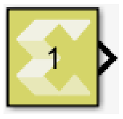
The Conjugate block has no parameters to set.

Constant

Provides constant value as a source.

Library

Source



Description

The Constant block generates a constant output of the value specified by the **Constant value** parameter. If you select **Interpret vector parameters as 1-D** parameter and specify the constant value as a scalar, row matrix, or column matrix, then the output is a 1-D array. Otherwise, the output is always two-dimensional. The Constant block supports real or complex constant values.

Data Type Support

By default, the Constant block outputs a signal with **double** data type and the same complexity as the **Constant value** parameter. However, you can specify the output to be any data type that Model Composer supports, including fixed-point and half data types by selecting the **Output data type** parameter.

Parameters

- **Constant value:**

  The Constant value parameter specifies the constant value output of the block.

  You can enter any expression that MATLAB® evaluates as a scalar or matrix.

  Table 68: **Settings**

Choices	Description
1.0	Constant Value

- **Interpret vector parameters as 1-D:**

  Specifies whether the constant value should be interpreted as a 1-D array.

  Table 69: **Settings**

Choices	Description
On	If the specified constant value is a scalar, row matrix, or column matrix, then the output is a 1-D array. Otherwise, the output is a 2-D matrix.
Off	The output is a 2-D scalar or matrix.

- **Sample time:**

  Specifies block sample time as a numerical value. The sample time of a block indicates when, during simulation, the block generates outputs or updates its internal state.

The block allows you to specify a block sample time directly as a numerical value. For example, to generate output at every two seconds, you can directly set the discrete sample time by specifying the numerical value of 2 as the **Sample time** parameter.

Settings for **Sample time** are:

Table 70: **Sample Time Parameter**

Sample Time Type	Sample Time Supported	Description	Supported?
Discrete	T_S	Generates output at discrete samples T_S. Discrete sample time is supported with the initial time offset value fixed to 0. The initial offset value is not configurable.	Yes
Continuous	0	Generate output continuously by dividing the sample hits into major time steps and minor time steps. The Simulink ODE solver you choose integrates all continuous states from the simulation start time to a given major or minor time step.	No
Inherited	-1	The sample time value is inherited from other sources. It is determined by applying a set of heuristics and based on the context of the block within the model by Simulink. Allowing a design to inherit sample time maximizes its reuse potential. For example, a design might fix the data types and dimensions of all its input and output signals. But you could reuse the design with different sample times (for example, discrete at 0.1 or discrete at 0.2, triggered, and so on).	Yes
Constant	inf	Constant sample time. Same as inherited sample time for HLS blocks.	Yes
Variable	-2	Variable Sample time.	No
Triggered	-1 (implicit)	Execute the block upon some implicit condition when it is inside a subsystem like triggered, function call, or iterator subsystem.	Yes

For additional details for simulating sample time, see Types of Sample time in the Simulink documentation.

- **Output data type:**

  This parameter specifies the data type of the output signal.

  If the output data type is one of the integer types, then the Constant value is rounded off as explained below.

  A value with the fractional part less than 0.5 is rounded towards zero, the fractional part more than 0.5 is rounded away from zero.

  In case of a tie (fractional part is 0.5), the Constant value is rounded up, i.e. the negative Constant value, is rounded towards zero and the positive Constant value is rounded away from zero.

- **Settings:**

  The following data types are supported:

Table 71: **Output Data Type Parameter**

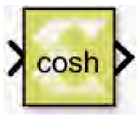
Setting	Description
double, single, and half	Floating point data types.
int8, uint8, int16, uint16, int32, uint32	Signed and unsigned integer data types.
Logical	Boolean data type.
Fixed point data type	• Fixed-point arithmetic data type with configurable output data type attributes like signedness, word length, fractional length. • Constant value conversion attributes rounding and overflow for reading constant value parameter.
Data type Expression	A string that specifies the output data type. See "Working with Data Type Expression" in the *Vitis Model Composer User Guide* (UG1483).

cosh

Element-wise computation of the hyperbolic cosine for a given argument

Library

Math Functions / Math Operations



Description

The cosh block returns the output of the function cosh(x), which is the hyperbolic cosine, for each element in array x.

The hyperbolic cosine of `x` is:

$$\cosh(x) = \frac{e^{x} + e^{-x}}{2}$$

Data Type Support

Data type support is:

- Dimension: Input can be scalar, vector, or matrix.
- Data Types: Input supports signals of integer type, floating point data types (double, single, and half) and fixed point type.
- Complex Numbers: Complex numbers are not supported.

Output has the same dimension and data type as the input.

Parameters

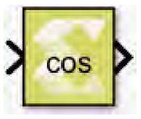
The cosh block has no parameters to set.

Cosine

Computes cosine value for the input.

Library

Math Functions / Math Operations



Description

The Cosine block returns the output of the function cos(x) for each element in array x.

Data Type Support

Data type support is:

- Dimension: Input can be scalar, vector, or matrix.

- Data Types: Input supports signal of floating-point data types (double, single, and half), and signed fixed point type. It does not support integer, Boolean, and unsigned fixed point data types.
- Complex Numbers: Complex numbers are not supported.

Output has the same dimension and data type as the input. However, if the data type of the input is a fixed point type, the data type of the output is fixed point type with integer width fixed as 2.

Parameters

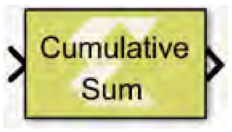
The Cosine block has no parameters to set.

Cumulative Sum

Compute the cumulative sum along the specified dimension of the input

Library

Math Functions / Math Operations



Description

The Cumulative Sum block computes the cumulative sum of the input signal along the specified dimension or across time (running sum). The output signal has the same dimensions, data type and complexity as the input signal.

Summing is performed in this way:

- Summing along Rows: If the block is configured for cumulative sum along rows, each element in the output signal is the sum of the corresponding element in the input and all of the elements in the same row and to the left of that element. If the input is 1-dimensional, each element in the output signal is the sum of the corresponding element in the input and all of the preceding elements.
- Summing along Columns: If the block is configured for cumulative sum along columns, each element in the output signal is the sum of the corresponding element in the input and all of the elements in the same column and above of that element. If the input is 1-dimensional, each element in the output signal is the sum of the corresponding element in the input and all of the preceding elements.

- Running sum: If the block is configured for running sum, each element in the output signal is the sum of the corresponding element in the input signal over time. In this case you can specify an optional reset port and restart the running sum when the reset signal is asserted.

Data Type Support

Data type support is:

- Data input: The data In input is the data signal to be summed. It supports integer, fixed point and floating-point data types but not boolean. The signal can be complex or real. The signal can be a scalar, vector or matrix.
- Reset: The Reset input is applicable only for running sum with non-default reset type. The reset signal must be scalar and real, and its data type must be integer or floating-point. Fixed point data type is not supported.
- Output: The data type, dimension and complexity of the output signal are the same as those of input signal.

Parameters

Sum input along

This parameter specifies the dimension along which sum elements are computed.

Settings for the **Sum input along** parameter are:

Table 72: **Sum Input Along Parameter**

Setting	Description
Columns	The block computes the cumulative sum of each column of the input.
Rows	The block computes the cumulative sum of each row of the input.
Channels (running sum)	The block computes a running sum for each element of the input across time. When you select the **Channels (running sum)** option, you will also have to specify a **Reset port** parameter.

Reset port

This parameter applies only to running sum. The **Reset port** parameter appears if you select **Channels (running sum)** for the **Sum input along** parameter.

Settings for the **Reset port** parameter are:

Table 73: **Reset Port Paremeter**

Setting	Description
None	Omits the Reset port.

Table 73: **Reset Port Paremeter** *(cont'd)*

Setting	Description
Non-zero sample	Triggers a reset operation at each sample time that the Reset input is not zero.

Data Type Conversion

Convert the input to the data type of the output.

The block warns or errors out when an integer or fixed-point output overflows during simulation. To configure, select **Configuration Parameters→Diagnostics→Data Validity**. In the Data Validity pane, set **Wrap** or **Saturate** to Overflow.

Library

Signal Attributes

Description

The Data Type Conversion block has one input and one output. It converts the value of the input signal to the data type of the output. This conversion tries to preserve the mathematical value of the input signal. The data type of the output is specified via the mask dialog. The conversion is governed by the following rules:

- Conversions where the output data type is fixed-point, first select the nearest number that can be represented, taking into account the overflow mode. In case of a tie, the rounding mode breaks the tie.
- Conversions where the output data type is integer are performed as in the C language. Overflow is handled via truncation.
    - As per IEEE Standard for Floating-point Arithmetic (IEEE Standard 754, Section-7.2), conversion from floating-point to integral is an invalid operation, when the floating-point value is outside the range of the destination integer data type. In this case, the output integer value depends upon the implementation of a C compiler. Hence, the results from the HLS Data Type Conversion block may differ from the results from Simulink® Data Type Conversion block.
    - When the floating-point input value is outside the range of the integer data type, the simulation results between Model Composer and RTL co-simulation in Vitis HLS may also differ.

   - During simulation, to check whether the input floating-point value goes outside the range of the destination integer type, in Simulink select **Model Configuration Parameters → Diagnostics → Data Validity** . Then set **Saturate on overflow** to either **warning** or **error**.
- Conversions where the output data type is floating point follow the rules implemented in the C language.

Data Type Support

The input signal can be double, single, an integer, boolean, Xilinx supported half or Xilinx supported fixed-point data type.

The data type of the output is specified the mask parameters.

The input can be real or complex, and scalar, vector, or matrix. The output signal has the same complexity and dimensions as the input signal.

Parameters

- **Output data type:**

  This parameter specifies the data type of the output signal. If `fixed` is specified, more parameters are available.

  Settings for the **Output data type** parameter are as follows.

Table 74: **Output Data Type Parameter**

Setting	Description
double	double precision floating point
single	single precision floating point
int8	8-bit signed integer
uint8	8-bit unsigned integer
int16	16-bit signed integer
uint16	16-bit unsigned integer
int32	32-bit signed integer
uint32	32-bit unsigned integer
logical	boolean
fixed	Xilinx supported fixed-point
half	Xilinx supported half precision floating point
data type expression	A string that specifies the output data type. See "Working with Data Type Expression" in the *Vitis Model Composer User Guide* (UG1483).

- **Signedness:**

  If the **Output data type** is set to **fixed**, the **Signedness** parameter specifies whether the output is a signed fixed-point or unsigned fixed-point data type.

Settings for the **Signedness** parameter are as follows.

Table 75: **Signedness Parameter**

Setting	Description
Signed	The output type contains both positive and negative numbers.
Unsigned	The output type contains only non-negative numbers.

This parameter is available only if **fixed** is selected as the setting for parameter **Output data type**.

- **Word length:**

  If the **Output data type** is set to **fixed**, the **Word length** parameter specifies the number of bits used to represent it.

Table 76: **Word Length Parameter**

Choices	Description
16	
N	A positive integer

This parameter is available only if **fixed** is selected as the setting for parameter **Output data type**.

- **Fractional length:**

  If the **Output data type** is set to `fixed`, the **Fractional length** parameter specifies the number of bits to the right of the binary point.

Table 77: **Fractional Length Parameter**

Choices	Description
10	
N	An integer

This parameter is available only if **fixed** is selected as the setting for parameter **Output data type**.

- **Round:**

  If the **Output data type** is set to **fixed**, the **Round** parameter allows you to select among five rounding and two truncation options.

If one of the five rounding options is selected, the block always rounds to the nearest *supported precision*. The five rounding choices are relevant only in case of a tie. For example, assume the output type is signed fixed-point, with a word length of 6 and a fractional length of 2, and the input to the block is 2.74. In this case, the output is rounded to the nearest supported precision, 2.75, regardless of which one of the five rounding modes is selected. However, if the input value is 2.625 (halfway between 2.5 and 2.75), then the output value depends on the chosen rounding mode. If **Round to plus infinity** is selected, the value will be 2.75, and if **Round to zero** is selected, the value will be 2.5. For more information on this, refer to the *Vitis High-Level Synthesis User Guide* (UG1399).

If one of the two truncation options is selected, the output will be truncated to the supported precision specified by the truncation selection.

Truncation to minus infinity is the default setting for **Round** and requires the smallest hardware resources among all the options.

The **Round** parameter is available only if **fixed** is selected as the setting for the **Output data type** parameter.

Settings for the **Round** parameter are:

Table 78: **Round Parameter**

Setting	Description
Round to plus infinity	Rounding to plus infinity
Round to zero	Rounding to zero
Round to minus infinity	Rounding to minus infinity
Round to infinity	Rounding to infinity
Convergent rounding	Convergent rounding
Truncation to minus infinity	Truncation to minus infinity
Truncation to zero	Truncation to zero

- **Overflow:**

  If the **Output data type** is set to `fixed`, the **Overflow** parameter specifies the overflow mode applied during conversion.

  This parameter is available only if **fixed** is selected as the setting for parameter **Output data type**.

  Settings for the **Overflow** parameter are:

Table 79: **Overflow Parameter**

Setting	Description
Saturation	Saturation

Table 79: **Overflow Parameter** *(cont'd)*

Setting	Description
Saturation to Zero	Saturation to zero
Symmetrical Saturation	Symmetrical saturation
Wrap around	Wrap around
Sign-Magnitude Wrap Around	Sign magnitude wrap around

- **Type Expression:**

  If the **Output data type** is set to **data type expression**, the **Type Expression** parameter specifies the output data type as a string.

  This parameter is available only if **data type expression** is selected as the setting for parameter **Output data type**.

- **Saturate on integer overflow:**

  This parameter specifies whether integer overflow is handled by wrapping (default) or by saturating. This parameter is relevant only if the output is integral (int8, int16, int32, uint8, uint16, uint32).

  When overflow is detected, the Diagnostic Viewer displays messages depending on the diagnostic action selected in the Configuration Parameters dialog box. To configure, in the **Configuration Parameters → Diagnostics → Data Validity** pane, set the Wrap or Saturate on overflow.

  Settings for the **Saturate on integer overflow** parameter are:

  Table 80: **Saturate On Integer Overflow Parameter**

Setting	Description
Not selected	Integer overflow is handled by wrapping.
Selected	Integer overflow is handled by saturation.

If the **Output data type** is set to **fixed** and overflow is detected, the Diagnostic Viewer displays messages that depend on the diagnostic action you specify in the Simulink Editor. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink Editor, then set the **Wrap on overflow** or **Saturate on overflow** parameter.

Delay

Delay input signal by specified number of samples

Library

Signal Operations

Description

The delay block produces an output signal by delaying the input signal by the number of samples specified in the block dialog box. If the latency of the block is N, then the N-1 first output samples are always 0, and the N-th output sample is the first input sample.

Data Type Support

All data types are supported.

Input can be a vector or a matrix. If input is a vector or a matrix and the latency value is a scalar, the scalar value will apply to all the elements of the input.

Output is complex if the input is complex.

Parameters

Latency

The **Latency** parameter specifies the number of samples by which the input signal is delayed.

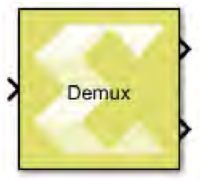
Latency should be a real, non-negative, scalar integer with minimum value as 1 and maximum value as 2^25.

Demux

Separates a vector input into a number of scalar and vector outputs.

Library

Signal Routing



Description

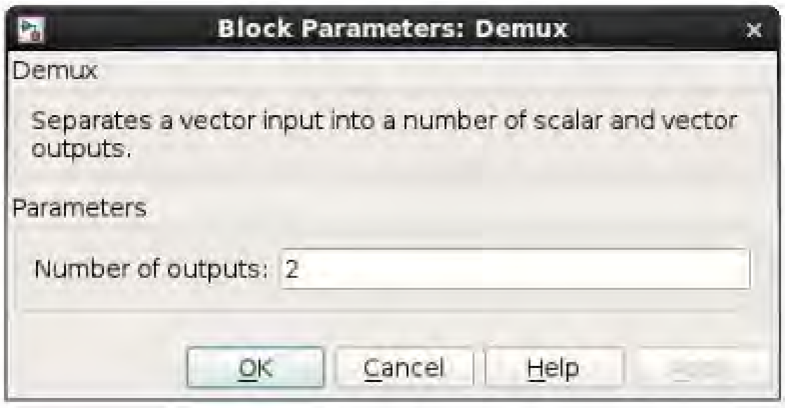
The Demux block input signal can be a scalar, a vector, a row matrix (1xN), or a column matrix (Nx1). The block splits elements in the input signal into scalar and vector type output signals according to the width of each output port starting from the first port at the top right.

The number of output ports and the port widths are configurable using the Number of outputs block parameter,

You can specify the width of each port or it can be dynamically computed by the block based on how the parameter value is provided. Refer to the Parameters section below for more details.

When the value of the Number of outputs parameter is changed, the output ports are either added or removed starting from the last port at the bottom right.

Figure 392: **Demux Parameter Dialog**


Block Parameters: Demux

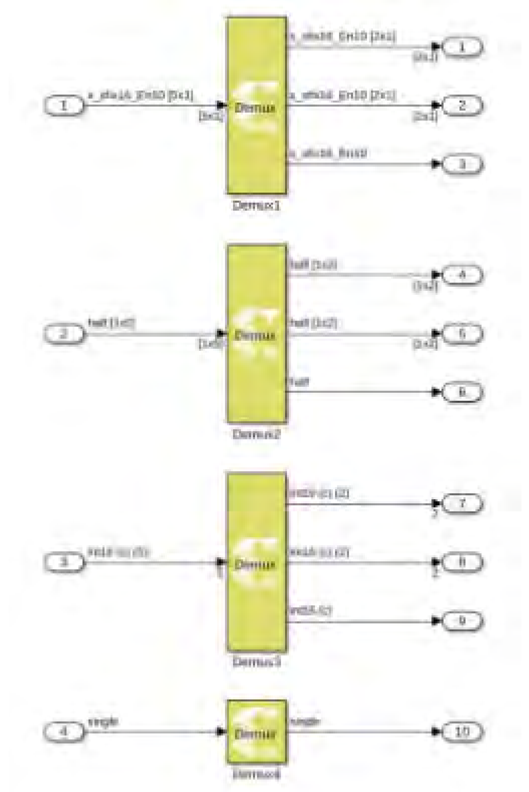
Demux

Separates a vector input into a number of scalar and vector outputs.

Parameters

Number of outputs: 2

Figure 393: **Demux Diagaram**



The output ports are added/removed starting from the last port at the bottom right.

Data Type Support

- **Inputs:**
  - The block has one input port.
  - The input signal can be a scalar, a vector (N), a row matrix (1xN), or column matrix (Nx1), where N is the width of input signal.
  - The Demux block supports all native data types (double, single, uint8, int8, uint16, int16, uint32, int32, and boolean), and Model Composer supported half and fixed-point data types.
  - The block supports input data in real or complex numeric type.
- **Outputs:**
  - The number of outputs for the block is specified using the Number of outputs block parameter.
  - The value of the block parameter can be a finite positive integer, P, or an array of integers. The numbers in the array are used to decide the number of outputs as well as the width of each output signal.

- An output signal can be configured as a scalar, a vector (M), a row matrix (1xM), or a column matrix (Mx1) where M is less than or equal to the width of the input signal.
- The sum of widths of all output signals is ensured to be the same as the input signal width.
- The data type and the numeric type (real or complex) of output signals are the same as those of the input signal.

Parameters

- **Number of outputs:**

  This parameter takes number of outputs in several ways. Depending upon the parameter value, the output ports are added/removed starting from the last port at the bottom right.

Table 81: **Number of Outputs**

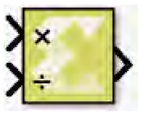
Option	Choices	Description
1	2	The block icon is initially created with two output ports. The input signal width is equally divided between the two outputs. If the input signal width is an odd number, then any remainder of the width is assigned to the first port at the top right.
2	P	A finite integer value representing the number of output ports. P must be greater than 0. The block icon is redrawn with the specified number of output ports. The widths of the output ports are dynamically computed by the block as follows: The width of the input is equally divided among the outputs. Any remainder of the width is assigned, one each, to the outputs starting from the first port at the top right. For example, if N is 3, and the width of the input is 14, then the first output is assigned with the first 5 input elements, the second output is assigned with the next 5 input elements, and the third output is assigned with the last 4 input elements.
3	[P]	A finite positive integer in square brackets is treated just like option 2 above. Here, the number of outputs will be P.
4	[-1 -1 -1]	The block icon is redrawn with 3 output ports. Here -1 means that the width of the particular output port needs to be computed in the same way as it is explained in the option 2 above.
5	[3 -1 -1]	The block icon is redrawn with 3 output ports. You specify the width of the first output, and Model Composer computes the widths of the second and the third outputs. For example, if width of the input is 8, and the first output width is 3, then the remaining width of 5 is divided between the second and the third outputs. This results in the widths of the second and the third outputs to be set to 3 and 2 respectively.
6	3 3 1	The block icon is redrawn with 3 output ports. The width of each output port is already specified by the user. The sum of the width of the outputs is 7. The width of the input must be 7, otherwise, an error message appears.

Divide

Element-wise division

Library

Math Functions / Math Operations



Description

The Divide block has two input ports and one output port. The output signal (quotient) is the result of element-wise division of the first input (dividend) by the second input (divisor).

Data Type Support

The data types of the dividend and divisor can be any integer, fixed point or floating point data type. Boolean dividends or divisors are not supported. The input signals can be real or complex. The input signals can be scalars, vectors or matrices. If neither input signal is scalar, they must either be vectors of the same length or matrices with the same number of rows and columns. When the output is integer or fixed point data type, the result is truncated to zero.

The output data type is chosen for maximal alignment with Vitis HLS. If the dividend and divisor are fixed point types and the fixed point parameters of the dividend are W1 (word length), FW1 (fractional length), and S1 (signedness), and the fixed point parameters of the divisor are W2, FW2, and S2, then the fixed point parameters of the quotient are as follows:

- S = S1 || S2
- FW2 ≥ 0
    - W = W1 + FW2 + S2
    - FW = FW1
- FW2 < 0
    - W = W1 +S2
    - FW = FW1 + FW2

Parameters

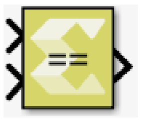
The Divide block has no parameters.

Equals

Perform element-wise equal to relational operation on the inputs. The top input corresponds to the first operand.

Library

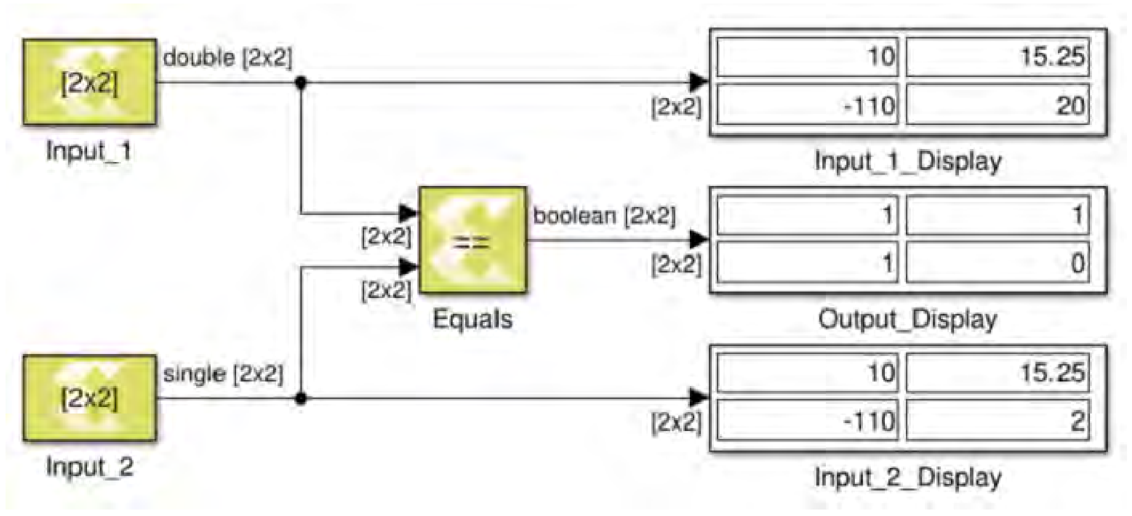
Relational Operations



Description

The Equals block performs element-wise comparison of inputs for equality. The block has two input ports and one output port. The output is true if the first input is equal to the second input.

Figure 394: **Equals Block**



Data Type Support

Data types accepted at the inputs of the block are:

- The block supports all data types supported by Vitis Model Composer.
- The block supports inputs having different data types. The output data type is always Boolean.
- The block supports mixed dimensions for inputs, where one input is a scalar, and the other input is a vector/matrix. The scalar value is compared with each element of the multi-dimensional input value for equality. The output has the same dimension as the multi-dimension input.
- If both inputs are non-scalar then their dimensions must match.

The output data type is always Boolean.

Parameters

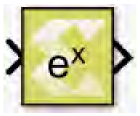
The Equals block has no parameters to set.

Exp

Perform an element-wise exponential value of the input

Library

Math Functions / Math Operations



Description

The Exp block returns the exponential e^x for each element in array `x`. The block supports all data types except Boolean. The input can be scalar, vector or matrix.

Data Type Support

Data types accepted at the inputs of the block are:

- Dimension : Input can be scalar, vector, or matrix.
- Data Types: Input supports signal of integer, fixed point, and floating point data type. It does not support Boolean inputs.
- Complex numbers are not supported.

The output has the same dimension and type as the input.

Parameters

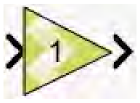
The Exponential block has no parameters to set.

Gain

Element-wise multiplication of the input by a constant gain factor

Library

Math Functions / Math Operations



Description

The Gain block multiplies the input signal by a constant gain factor.

You can specify the data type of the gain constant and built-in type promotion rules apply to determine the output data type. Alternatively, the output data type can be made the same as the input type. In case of integer overflow the block supports the option to saturate output values at the output type limits.

The block warns or errors out when an integer output overflows during simulation. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink Editor, then set the **Wrap on overflow** or **Saturate on overflow** parameter.

Data Type Support

The block supports all data types except Boolean.

Data type support for the block is:

- The input can be a scalar, vector or matrix.
- If the input is a vector or matrix and the **Gain** is a scalar, the scalar value will apply to all the elements of the input.
- If neither the input nor the gain constant are scalar the dimensions of the input and the gain constant must match.

The output is complex if either the **Gain** constant or the input is complex.

Parameters

Gain

Specifies the constant gain factor. The **Gain** can be any valid MATLAB expression that evaluates to a real or complex scalar, vector, or matrix.

Gain data type

This parameter specifies the type of conversion to be applied to the **Gain** factor constant before multiplication. If **fixed** is specified more parameters are available.

Settings for the **Gain data type** parameter are:

Table 82: **Gain Data Type Parameter**

Setting	Description
double	double precision floating-point
single	single precision floating-point
int8	8-bit signed integer
uint8	8-bit unsigned integer
int16	16-bit signed integer
uint16	16-bit unsigned integer
int32	32-bit signed integer
uint32	32-bit unsigned integer
fixed	fixed-point
half	half precision floating-point
data type expression	A string that specifies the output data type. See "Working with Data Type Expression" in the *Vitis Model Composer User Guide* (UG1483).

Unless the **Output data type same as input** parameter is enabled, the output data type will be a function of the input type and the specified **Gain data type**.

- If either input or gain types are floating-point (double, single, or half), the output type will be floating-point. If both are floating-point, the output type will be the larger of both. The smaller type will be promoted to the larger before the operation.
- Otherwise, if either of input or **Gain data type** are fixed-point, the output type will be fixed-point with a bit width sufficient to hold the full output result. The other type (input or **Gain data type**) will be promoted to its equivalent fixed-point type.
- Otherwise, the input and **Gain data type** are integers. The output type will be the larger of either input or Gain data type, and will be a signed integer if either one is signed.

Output data type same as input

This parameter specifies the way the output data type is determined.

If disabled (unchecked) the output type is computed via built-in type promotion rules. If enabled (checked), the output data type is the same as the input type.

Saturate on integer overflow

This parameter specifies the behavior in case of integer overflow. By default the option is disabled and overflow would result in value wrap. With the option enabled, integer overflow gets mitigated by saturation at the limits of the output data type.

Settings for the **Saturate on integer overflow** parameter are:

Table 83: **Saturate On Integer Overflow Parameter**

Setting	Description
Unchecked	Wrap around
Checked	Saturation

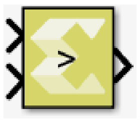
When overflow is detected, the Diagnostic Viewer displays messages that depend on the diagnostic action you specify in the Simulink Editor. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink Editor, then set the **Wrap on overflow** or **Saturate on overflow** parameter.

Greater

Performs element-wise greater than relational operation on the inputs. The top input corresponds to the first operand.

Library

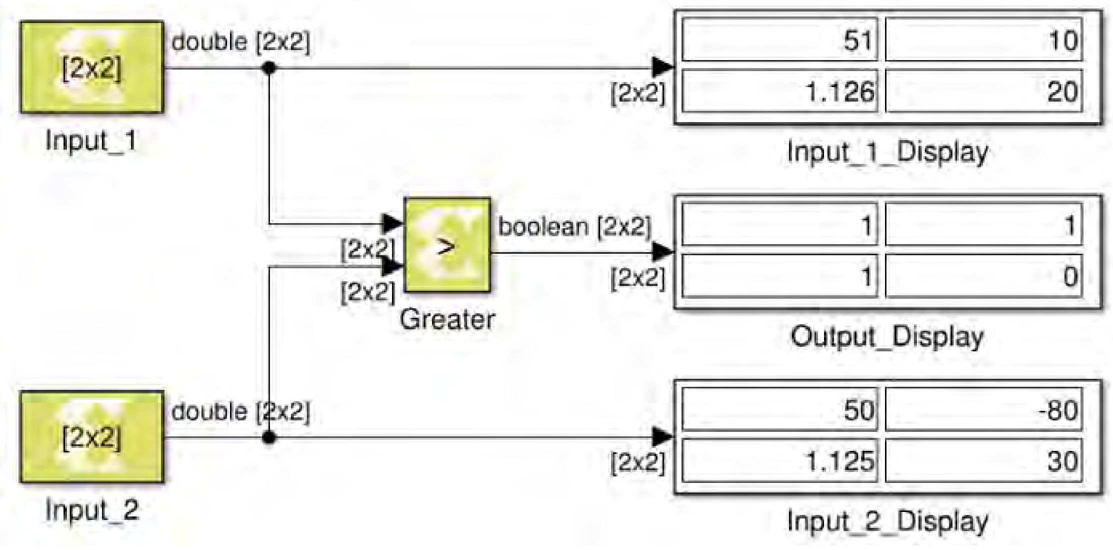
Relational Operations



Description

The Greater block has two input signals and one output signal. The block compares the two inputs using element-wise greater than relational operation. The first input corresponds to the top input port and the second input to the bottom input port. The dimension of the output signal matches the dimensions of the input signals. An element of the output signal is true if the corresponding element of the first input signal is greater than the corresponding element of the second signal. Otherwise the element is false.

Figure 395: **Greater Block**



Data Type Support

Data type support for the Greater block is:

- The data types of the input signals can be integer, fixed-point, Boolean, or floating-point data type.
- The input signals can be a scalars, vectors, or matrices. If both inputs are not scalar, their dimensions must match.
- The input signals must be real.
- The output signal is Boolean.
- The dimension of the output signal is scalar if both inputs are scalar. Otherwise it matches the dimensions of the non-scalar input.

Parameters

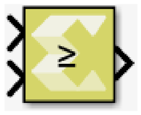
The Greater block has no parameters to set.

Greater Equals

Perform element-wise greater than or equal relational operation on the inputs. The top input corresponds to the first operand.

Library

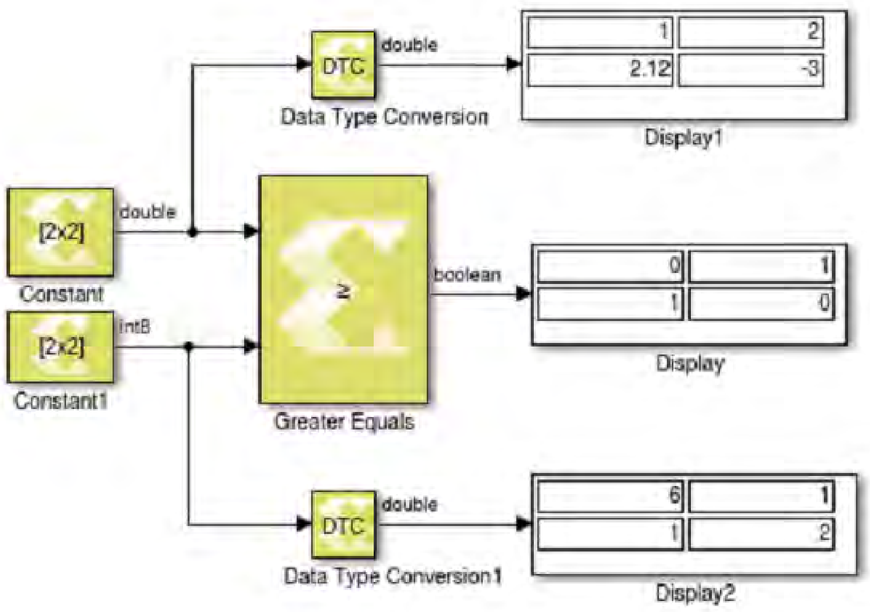
Relational Operations



Description

The Greater Equals block performs element-wise greater than or equal relational operation on the inputs. The upper input is the first input and the lower input is the second input. The block returns true if the first input is greater than or equal to the second input. The output equals 1 for true and 0 for false.

Figure 396: **Greater Equals Block**



Data Type Support

Data types accepted at the inputs of the block are:

- Data Types: Greater Equal block supports all data types supported by Model Composer (integer, floating-point, fixed-point, and Boolean).
- Dimension: The inputs can be scalar, vector, or matrix, or a combination of scalar, and matrix, or vector. If both the inputs are matrix or vector, they should have same dimension.
- Complex Number Support: No

The output is always Boolean.

Outputs for the different input types are:

Table 84: **Input/Output**

Inputs	Output
Both are scalar	Scalar
Both are vector	Vector of same dimension
Both are matrix	Matrix of same dimension
One is scalar and the other is vector or matrix	Dimension is that of the vector or matrix

Parameters

The Greater Equals block has no parameters to set.

Hermitian

Perform element-wise conjugate transpose operation on the input signal.

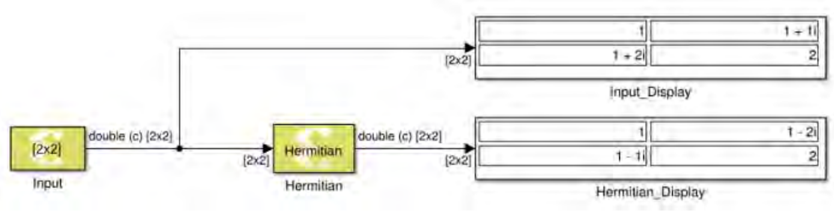
Library

Math Functions / Matrices and Linear Algebra

Description

The Hermitian block performs a conjugate transpose operation on the input signal.

Figure 397: **Hermitian Block**



Data Type Support

This block supports all data types supported by Xilinx® Model Composer. The input signal can be real or a complex number of scalar, vector, or matrix type.

Parameters

The Hermitian block has no parameters to set.

Interface Spec

Specify the RTL interfaces for a subsystem

Library

Tools

Description

The Interface Spec block allows you to control what RTL interfaces should be synthesized for the ports of the subsystem in which the Interface Spec block is instantiated. This affects only code generation and synthesis, when an RTL model (an IP) is synthesized by Vitis HLS from the C++ model produced by Model Composer. The block has no effect on Simulink® simulation of your design. If your design does not have an Interface Spec block, Model Composer selects default interfaces for you. Interface synthesis is supported only to the subsystem for which you are generating C++ code. Therefore any Interface Spec blocks instantiated in subsystems nested within the subsystem for which you are generating C++ code are ignored.

The Interface Spec block is used as follows:

1. Instantiate the Interface Spec block in the subsystem for which you want to generate C++ code. The **Input ports** tab will be populated with one row for each input port of the parent subsystem. Similarly, the **Output ports** tab has one row for each output port of the parent subsystem.
2. Fill out the **Function Protocol**, **Input ports**, and **Output ports** tabs.

The information gathered by the Interface Specification block consists of three parts:

- The block-level Interface Protocol. This protocol is used to tell the IP when to start processing data. It is also used by the IP to indicate whether it accepts new data, or whether it has completed an operation, or whether it is idle.
- The port-level Interface Protocol for each input port of the parent subsystem.
- The port-level interface protocol for each output port of the parent subsystem.

The choice of port-level interface protocol should take into account the following considerations:

- Large array or matrix ports should use a streaming protocol such as AXI4-Stream, FIFO, or AXI4-Stream (video).
- Scalar ports can be implemented using any of the following protocols: Default, AXI4-Lite Slave, Constant, Valid Port, No protocol
- Video signals can be transported over an AXI4-Stream (video) interface. In this case you also need to specify the video format YUV 4:2:2, YUV 4:4:4, RGB or Mono. For video formats that have more than 1 color component, you also need to specify which port carries which color component and you need to assign the same name for the 'bundle' attribute for these (3) ports. All of the ports (either 3 or 1) that make up the video signal are implemented by a single AXI4-Stream interface that include start-of frame and end-of-line sideband signals. This follows the specifications described in the *AXI4-Stream Video IP and System Design Guide* (UG934).
- An AXI4-Lite Slave interface allows you to implement one or more ports.
- For further details refer to Interface Synthesis in the *Vitis High-Level Synthesis User Guide* (UG1399).

The interface specification block currently supports subsystems with at most 8 input ports and 8 output ports.

Data Type Support

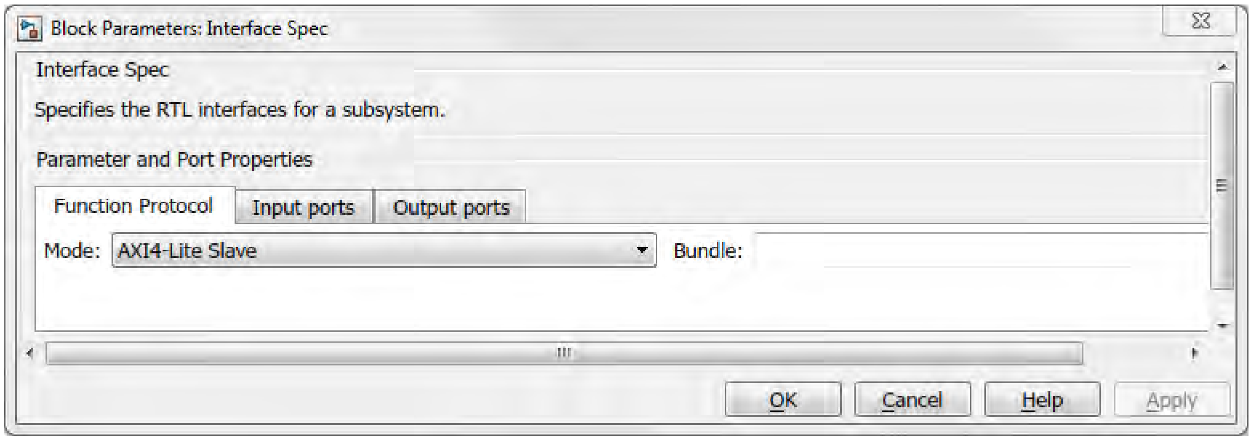
Data type support is not applicable to the Interface Spec block.

Parameters

The parameters for the Interface Specification block fall into the following groups.

- The parameters that apply to the function protocol. These are Mode, and Bundle. In the GUI dialog, these parameters appear in the 'Functional Protocol' tab.
- The parameters that apply to the Input ports. For each input port, there is 1 set of parameters Mode, Bundle, Offset, Video Format, and Video Component. In the Block Parameter dialog, these parameters appear in the 'Input ports' tab.
- The parameters that apply to the Output ports. For each output port, there is 1 set of parameters Mode, Bundle, Offset, Video Format, and Video Component. In the GUI dialog, these parameters appear in the 'Output ports' tab.

Figure 398: **Function Protocol Parameters**


Block Parameters: Interface Spec

Interface Spec

Specifies the RTL interfaces for a subsystem.

Parameter and Port Properties

Mode: AXI4-Lite Slave

Bundle:

Parameters on the **Function Protocol** tab are as follows:

- **Mode:**

  The **Mode** parameter specifies the block-level I/O protocol.

  Following are the settings for the **Mode** parameter.

  Table 85: **Mode Parameter**

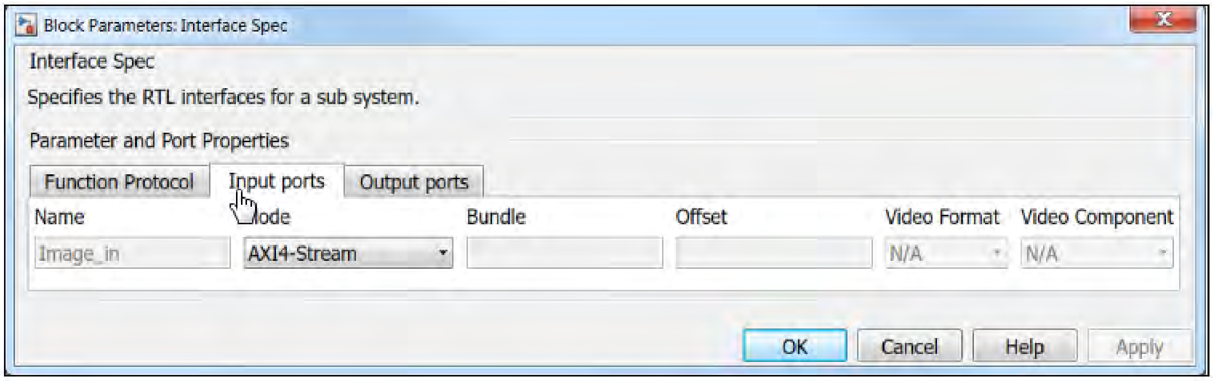
Setting	Description
AXI4-Lite Slave	Specifies AXI4-Lite Slave as the block-level I/O protocol.
Handshake	Specifies a handshake protocol as the block-level I/O protocol.
No block-level I/O Protocol	Specifies that there is no block-level I/O protocol.

  The default choice for the function protocol is 'AXI4-Lite Slave'. However, if the DUT does not have any scalar ports then Handshake is selected as default function protocol.

- **Bundle:**

  The **Bundle** parameter is used in conjunction with the AXI4-Lite Slave interface to indicate that multiple ports should be grouped into the same interface. Enter a legal identifier in the C language (cannot contain spaces or special characters) for **Bundle**.

Figure 399: **Input Ports Tab**



Block Parameters: Interface Spec

Interface Spec

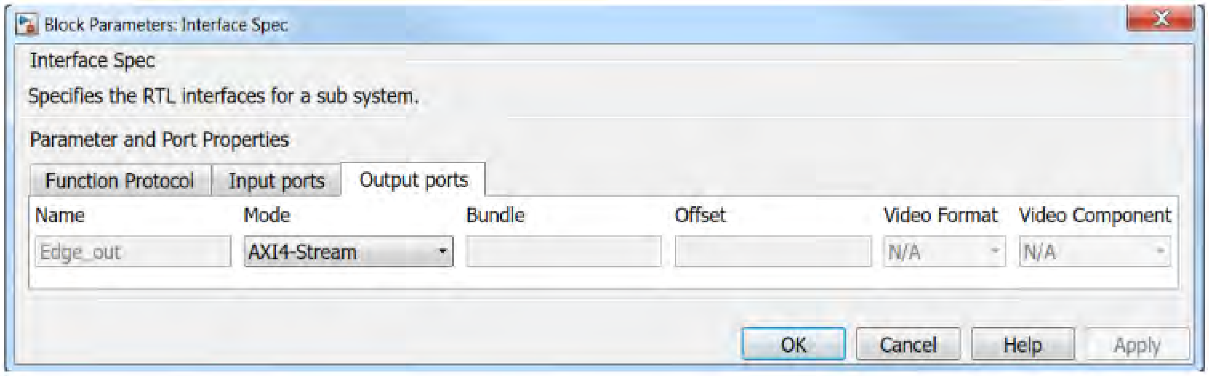
Specifies the RTL interfaces for a sub system.

Parameter and Port Properties

Input ports tab:

Name	Mode	Bundle	Offset	Video Format	Video Component
Image_in	AXI4-Stream			N/A	N/A

Figure 400: **Output Ports Tab**



Block Parameters: Interface Spec

Interface Spec

Specifies the RTL interfaces for a sub system.

Parameter and Port Properties

Output ports tab:

Name	Mode	Bundle	Offset	Video Format	Video Component
Edge_out	AXI4-Stream			N/A	N/A

Parameters on the **Input ports** and **Output ports** tabs are as follows.

- **Mode:**

  The **Mode** parameter specifies the I/O protocol for the input port or the output port.

  Settings for the **Mode** parameter are:

  Table 86: **Mode Parameter**

Setting	Description
Default	Specifies to use **AXI4-Lite Slave** if port is scalar, and use **AXI4-Stream** if the port is non-scalar.
AXI4-Stream	Specifies **AXI4-Stream** protocol.
AXI4-Stream (video)	Specifies **AXI4-Stream (video)** protocol. Allows you to specify **Bundle**, **Video Format**, and **Video Component** parameters.
AXI4-Lite Slave	Specifies **AXI4-Lite Slave** protocol. Allows you to specify **Bundle** and **Offset** parameters.
FIFO	Specifies a protocol for arrays whose elements are accessed in a sequential manner.

Table 86: **Mode Parameter** *(cont'd)*

Setting	Description
Valid port	Specifies a handshake protocol that only has a valid port.
Constant	Specifies a mode in which no I/O protocol is added to the port. The mode is intended for configuration inputs which only change when the device is in reset mode. This mode only applies to **Input ports**.
No protocol	Specifies that no I/O protocol is added to the port.
Block RAM	Specifies **Block RAM** interface protocol.

- **Bundle:**

  The **Bundle** parameter applies to the input ports or output ports and it is used in conjunction with the **AXI4-Stream (video)** interfaces that have more than one color component. In this case there should be one port for each color component and these ports should specify the same name for the **Bundle** attribute so that they will be grouped into the same **AXI4-Stream (video)** interface.

  The parameter is also used in conjunction with **AXI4-Lite Slave** interfaces to specify that ports with the same name for the **Bundle** attribute will be grouped into the same AXI4-Lite Slave interface.

  Enter a legal identifier in the C language (cannot contain spaces or special characters) for **Bundle**.

- **Offset:**

  The **Offset** parameter applies to the input ports or output ports and it is used in conjunction with the **AXI4-Lite Slave** interface. The parameter allows you to specify the address offset for a port within the **AXI4-Lite Slave** address map.

- **Video Format:**

  The **Video Format** parameter applies to the input ports or output ports and it specifies the color format for a video signal. It applies only to **AXI4-Stream (video)** interfaces. Options are **Mono**, **YUV 4:2:2**, **YUV 4:4:4**, and **RGB**.

- **Video Component:**

  The **VideoComponent** parameter applies to the input ports or output ports and it specifies the color component for a video signal. It applies only to **AXI4-Stream (video)** interfaces that use a **Video Format** with more than one color component. Options are **Mono**, **YUV 4:2:2**, **YUV 4:4:4**, and **RGB**.

  The **Video Component** selections for the different **Video Format** options are as follows.

Table 87: **Video Component Option**

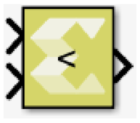
Video Format	Video Component Options
Mono	N/A
YUV 4:2:2	**Y**, **U**, **V**
YUV 4:4:4	**Y**, **U**, **V**
RGB	**R**, **G**, **B**

Lesser

Performs element-wise less than relational operation on the inputs. The top input corresponds to the first operand.

Library

Relational Operations



Description

The Lesser block has two input signals and one output signal. The block compares two inputs using element-wise lesser relational operation. The first input corresponds to the top input port and the second input to the bottom input port. The dimension of the output signal matches the dimensions of the input signals. An element of the output signal is true if the corresponding element of the first input signal is less than the corresponding element of the second signal; otherwise the element is false.

Data Type Support

Data types support for the Lesser block is:

- The data types of the input signals can be integer, fixed-point, boolean, or floating point data type.
- The input signals can be scalars, vectors, or matrices. If both inputs are not scalar, their dimensions must match.
- The input signals must be real.
- The output signal is boolean.
- The dimension of the output signal is scalar if both inputs are scalar. Otherwise it matches the dimensions of the non-scalar input.

Parameters

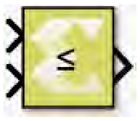
The Lesser block has no parameters to set.

Lesser Equals

Perform element-wise less than or equal relational operation on the inputs. The top input corresponds to the first operand.

Library

Relational Operations



Description

Performs element-wise less than or equal relational operation on the inputs. The block returns true (1) if the first input is less than or equal to the second input; returns false (0) otherwise.

Data Type Support

The Lesser Equals block supports inputs of native data types of MATLAB® and Ap_Fixed data type supported by Vitis HLS. The output type is always Boolean. It does not support complex data types. The block supports inputs of scalar, vector, and matrix dimensions. When both inputs have non-scalar dimensions then the dimensions of the inputs must match each other.

Parameters

The Lesser Equals block has no parameters to set.

Library Function

Import user created C function as a block

Library

Library can be specified after you create the Library Function block.

Description

The Library Function block allows you to bring C or C++ models into Model Composer for block simulation and code generation. The block I/O interface is determined by the function declaration, which is auto-discovered by the tool. The Library function block is created when you run the `xmcImportFunction` script to specify the library function sources files and header search paths.

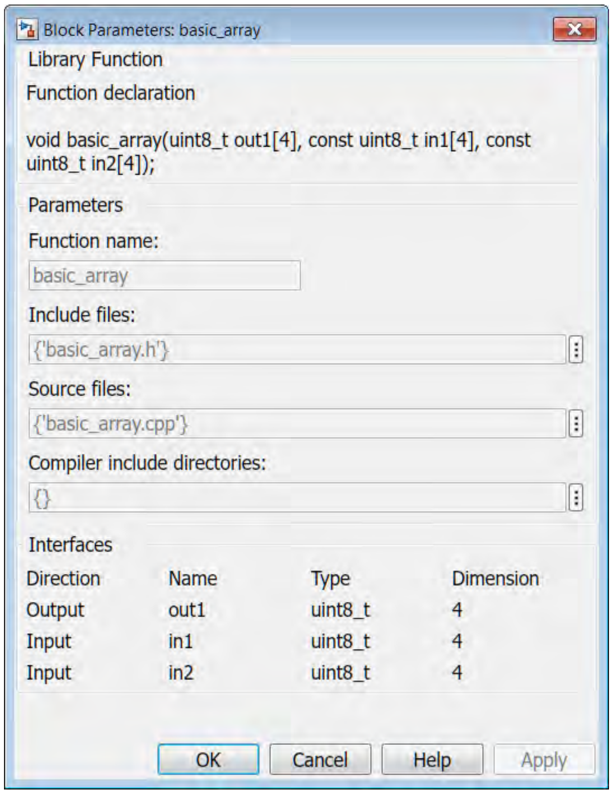
Data Type Support

You can import functions that have scalar, vectors, or matrices as function parameters. All data types, including fixed-point are supported. Complex values, real, and imaginary components, and phase angles are not supported.

Parameters

The block parameters dialog box for the Library Function block is shown below:

Figure 401: **Block Parameters**



Block Parameters: basic_array

Library Function

Function declaration

void basic_array(uint8_t out1[4], const uint8_t in1[4], const uint8_t in2[4]);

Parameters

Function name: basic_array

Include files: {'basic_array.h'}

Source files: {'basic_array.cpp'}

Compiler include directories: {}

Interfaces

Direction	Name	Type	Dimension
Output	out1	uint8_t	4
Input	in1	uint8_t	4
Input	in2	uint8_t	4

The dialog box indicates the settings for these parameters which were specified when the block was created:

- **Function name**

- **Include files**
- **Source files**
- **Compiler include directories**

You cannot change these settings from the dialog box. To change these settings, you will have to recreate the block using the `xmcImportFunction` command.

Log

Compute element-wise natural logarithm of input

Library

Math Functions / Math Operations

Description

The Log block returns the logarithm value for the provided input. The block supports input of all data types except Boolean. The input can be scalar, vector, or matrix.

Data Type Support

Data types accepted at the inputs of the block are:

- Data Types: Input supports signals of integer, fixed-point, and floating-point data type. The block does not support Boolean inputs.
- Complex Number Support: No

  The output has the same dimension and type as the input.

Parameters

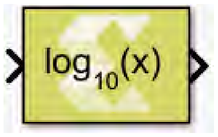
The Log block has no parameters to set.

Log10

Compute element-wise base 10 logarithm of input

Library

Math Functions / Math Operations



Description

The Log10 block returns the base 10 logarithm value for the input. The block supports input of all data types except Boolean. The input can be scalar, vector, or matrix.

Data Type Support

Data types accepted at the inputs of the block are:

- Data Types: Input supports signals of integer, fixed-point, and floating-point data type. The block does not support Boolean inputs.
- Complex Number Support: No

The output has the same dimension and type as the input.

Parameters

The Log10 block has no parameters to set.

Logical AND

Performs element-wise logical AND operation on inputs

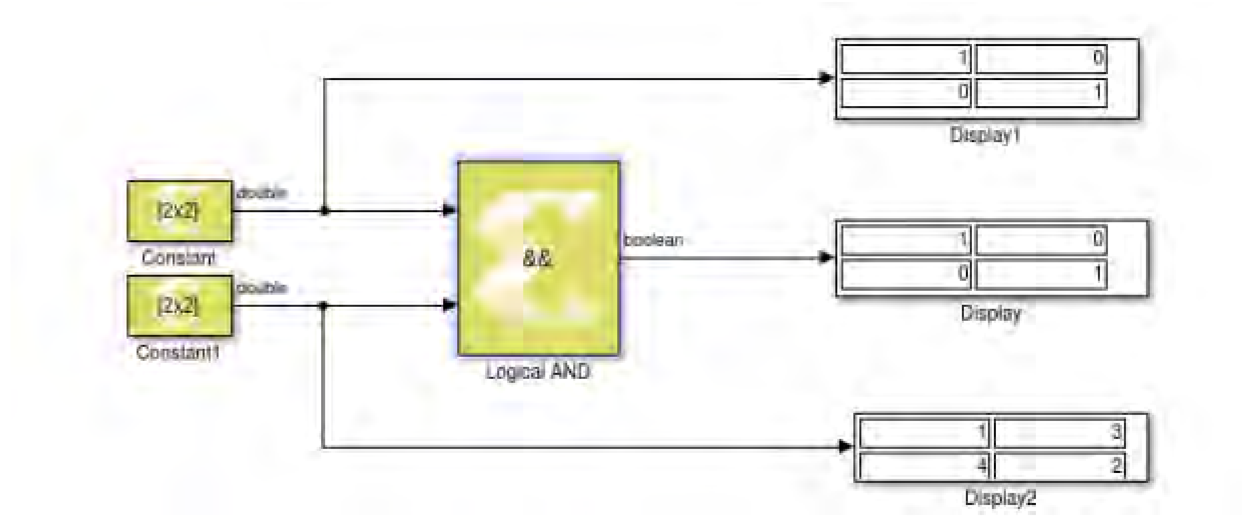
Library

Logic and Bit Operations

Description

The Logical AND block has two input ports and one output port. The block performs an element-wise logical AND operation on the inputs and produces a Boolean result on the output.

Figure 402: **Logical AND Block**



Data Type Support

- The block supports inputs of different data types. The output data type is always Boolean.
- The block only supports real inputs.
- If one input is non-scalar type then the other input can be scalar type.
- If both inputs are non-scalar type then their dimensions must match. In case of non-scalar inputs, the output has the same dimension as the inputs.

Parameters

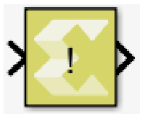
The Logical AND block has no parameters to set.

Logical NOT

Performs element-wise logical NOT operation on the input

Library

Logic and Bit Operations



Description

The Logical NOT block has one input port and one output port. The output is false if the input is a non-zero (true) value.

Data Type Support

- The Logical NOT block supports all data types supported by Model Composer.
- The block only supports real inputs.

Parameters

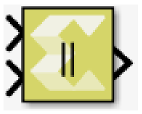
The Logical NOT block has no parameters to set.

Logical OR

Performs element-wise logical OR operation on inputs

Library

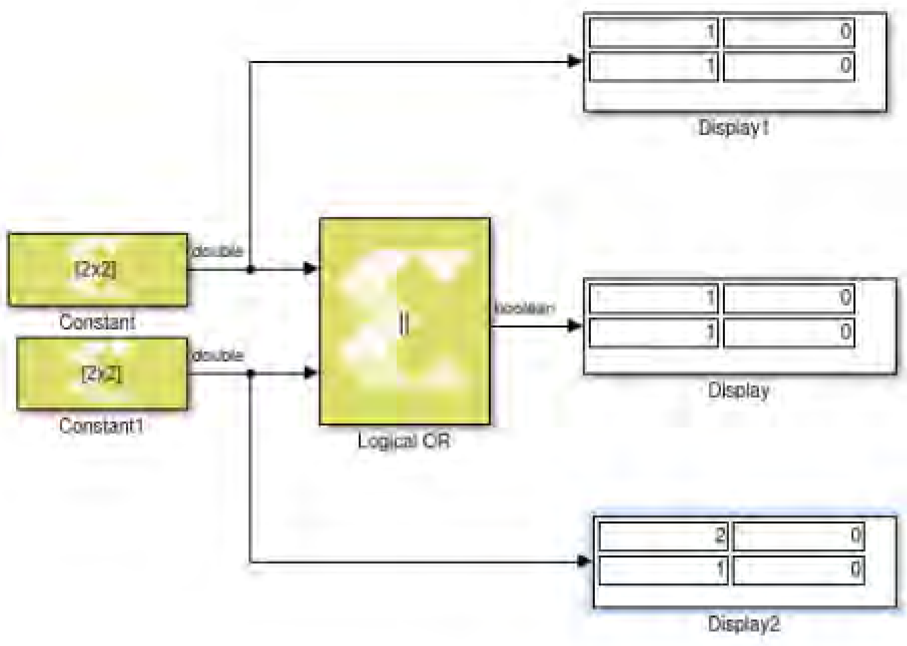
Logic and Bit Operations



Description

The Logical OR block has two input ports and one output port. The block performs an element-wise logical OR operation on the inputs and produces a Boolean result on the output.

Figure 403: **Logical OR Block**



Data Type Support

- The Logical OR block supports all data types supported by Model Composer.
- The block supports inputs of different data types. The output data type is always Boolean.
- The block only supports real inputs.
- The block supports scalar and non-scalar type inputs. If one input is non-scalar type then the other input can be scalar type. When both inputs are non-scalar type then their dimensions must match. In that case, the output has the same dimension as the inputs.

Parameters

The Logical OR block has no parameters to set.

Lookup Table

Perform one-dimensional lookup operation with an input index.

Library

Lookup Tables

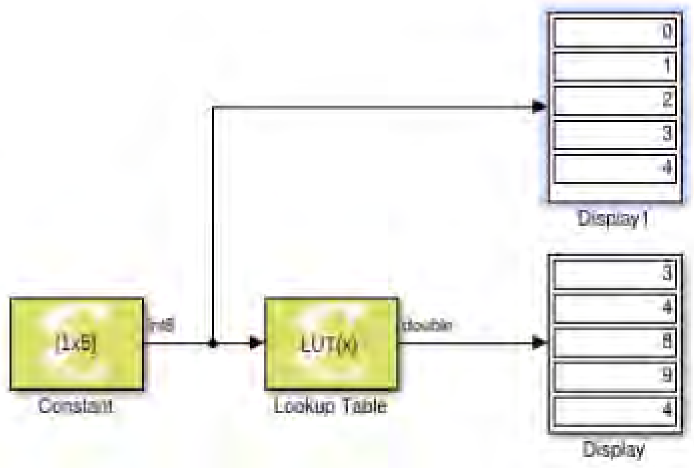
Description

The Lookup Table block implements a simple read-only memory block with an input index. The block maps input to an output value by looking up a table of values you define with a **Table data** parameter.

The input value is used as a zero-based index into the table data. The **Input bias** parameter is an offset to the index value (to support negative indexing). A **When input is out of range** parameter lets you specify the behavior of the block if the index value exceeds the valid table size range.

Note: If the table size is not an exact power of 2, the block incurs additional hardware cost when it is implemented in a Xilinx device, due to a remainder calculation on the index.

Figure 404: **Lookup Table Block**



In the example above, the **Table data** setting for the Lookup Table block is [7 3 4 8 9 4 1 5] with an input bias of 1.

Data Type Support

The Lookup Table block accepts the following data types to represent scalar index value: int8, uint8, int16, uint16, int32, uint32, and fixed-point type. Fixed-point inputs is shifted appropriately to generate an integer index.

The output data type is same as the **Table data** parameter type. Inputs for indexing must be real, but table data can be complex.

Parameters

Table Data

This parameter accepts a 1-D vector of table values. The size of the vector determines the valid index range for the input index. The data will be explicitly converted into the type specified in the **Output data type** parameter. If the input index exceeds table size and **Saturate at table ends** is specified for the **When input is out of range** parameter, then index value is saturated to either top or bottom of table size range. If **Wrap around** is specified for the **When input is out of range** parameter, the index is wrapped into the valid table size range.

Note: Large tables should be defined via a Simulink workspace variable due to space limitations in the block dialog box.

Input bias

This parameter is an offset into the table data that will be added to the index input. This makes it possible to use negative indices and perform look up operation.

When input is out of range

This parameter will guard the index value if it exceeds the valid table size range.

Following are the settings for the **When input is out of range** parameter.

Table 88: **When Input Is Out of Range Parameter**

Setting	Description
Saturate at table ends	If index value exceeds the valid table size range, then index value is saturated to either top or bottom of table size range, depending on the overflow direction.
Wrap around	If index value exceeds the valid table size range, then index value is wrapped into the valid table size range.

Output data type

Specifies the output data type.

Following are settings for the **Output data type** parameter.

Table 89: **Output Data Type Parameter**

Setting	Description
double	double precision floating-point
single	single precision floating-point
int8	8-bit signed integer
uint8	8-bit unsigned integer
int16	16-bit signed integer
uint16	16-bit unsigned integer

Table 89: **Output Data Type Parameter** *(cont'd)*

Setting	Description
int32	32-bit signed integer
uint32	32-bit unsigned integer
boolean	boolean
fixed	fixed-point
half	half precision floating-point
data type expression	A string that specifies the output data type. See "Working with Data Type Expression" in the *Vitis Model Composer User Guide* (UG1483).

Matrix Multiply

Compute matrix product of two input signals. The first operand is the top input on the block.

Library

Math Functions / Matrices and Linear Algebra

Description

The Matrix Multiply block has two input ports and one output port. The output signal is the matrix product of the input signals where the first operand corresponds to the top input.

Data Type Support

The data type of the input signals can be any floating-point, fixed-point, integer, or Boolean. The input signals can be real or complex. The input signals can be scalar, vector, or matrix, but they do need to be such that mathematically, the matrix product is defined. The table below shows valid combinations. Combinations that do not match any row in the table result in an error.

Table 90: **Data Type Combinations**

Dimensions of First Operand	Dimensions of Second Operand	Dimensions of Matrix Product	Conditions
K x L	L x M	K x M	K >= 1, L >= 1, M >= 1
K x L	L	K	K >= 1, L > 1
K x 1	1	K x 1	K >= 1
K	1	K	K >= 1
K	1 x M	K x M	K >= 1, M >= 1

The output data type is determined according to the following rules, in the order listed. T1 is a variable representing the type of the first operand; T2 is a variable representing the type of the second operand. These rules were chosen for maximum alignment with Vitis HLS, which may not correspond to the output data type computed via the internal rule of the Simulink® Matrix Product block.

Table 91: **Output Data Type**

Data Type of First Operand	Data Type of Second Operand	Data Type of Matrix Product
T1: floating-point	T2	The widest floating-point type between T1 and T2 if T2 is a floating-point type; otherwise T1
T1	T2: floating-point	The widest floating-point type between T1 and T2 if T1 is a floating-point type; otherwise T2
fixed-point	fixed-point	The smallest fixed-point type capable of representing the product without loss of precision
fixed-point	integer	The smallest fixed-point type capable of representing the product without loss of precision
integer	fixed-point	The smallest fixed-point type capable of representing the product without loss of precision
T1: integer	T2: integer	Let W1 be the bit width of T1 and W2 be the bit width of T2. The product is the integer type with bit width max (W1,W2) and it is signed if either T1 or T2 are signed.
boolean	T2	T2
T1	boolean	T1

Parameters

The Matrix Multiply block has no parameters to set.

Max

Outputs the maximum value of an input or element-wise maximum value of multiple inputs.

Library

Math Functions/Math Operations

Description

The Max block with a single scalar or vector input, outputs the maximum value of the input.

If the block has more than one input, the non-scalar inputs must have the same dimensions. Any scalar input is expanded to the dimensions of the non-scalar inputs, and the block outputs element-wise maximum value of the inputs.

Data Type Support

- The block supports floating-point, integer, and fixed-point data types.
- The block supports real valued inputs.
- For Number of Inputs = 1.
  - The block supports a scalar or vector (1-D or 2-D) input when it has only one input port. The output is a scalar.
- For Number of Inputs > 1.
  - The block supports scalar, vector, or matrix inputs when it has more than one input port.
  - The output has the same dimensions as those of the inputs when all the inputs have the same dimensions.
  - The block supports mixed dimensions for inputs when it has more than one input port, provided all the non-scalar inputs have the same dimensions. Any scalar input is expanded to the dimensions of non-scalar inputs, and the block outputs element-wise maximum value of the inputs. The output has the same dimensions as those of the non-scalar inputs.
- The block supports inputs having different data types. The output data type in this case is defined by the following set of rules.
  - If the data type of one of the inputs is a floating-point type, the data type of the output is the floating-point type among the data types of the inputs with the most precision.
  - If the data type of one of the inputs is a fixed-point type, the data type of the output is the smallest fixed-point type capable of representing the result without any loss of precision.
  - If the inputs are integral, the output is integral. If any input is signed, the output is signed. The bit width of the output is the largest among the bit widths of the inputs.

Parameters

- **Number of inputs:**

  This parameter determines the number of inputs.

Table 92: **Number of Inputs**

Choices	Description
1	Initially, the block icon has a single input.
N	A positive integer value. The block icon is redrawn with the specified number of input ports.

Min

Outputs the minimum value of an input or element-wise minimum value of multiple inputs.

Library

Math Functions/Math Operation

Description

The Min block with a single scalar or vector input outputs the minimum value of the input.

If the block has more than one input, the non-scalar inputs must have the same dimensions. Any scalar input is expanded to the dimensions of non-scalar inputs and the block outputs element-wise minimum value of the inputs.

Data Type Support

- The block supports floating-point, integer, and fixed-point data types.
- The block supports real valued inputs.
- Number of inputs = 1.
    - The block supports a scalar or vector (1-D or 2-D) input when it has only one input port. The output is a scalar.
- Number of inputs > 1.
    - The block supports scalar, vector, or matrix inputs when it has more than one input port.
    - The output has the same dimensions as those of the inputs when all the inputs have the same dimensions.
    - The block supports mixed dimensions for inputs when it has more than one input port, provided all the non-scalar inputs have the same dimensions. Any scalar input is expanded to the dimensions of non-scalar inputs and the block outputs element-wise minimum value of the inputs. The output has the same dimensions as those of the non-scalar inputs.
- The block supports inputs that have different data types. The output data type in this case is defined by the following set of rules.
    - If the data type of one of the inputs is a floating point type, the data type of the output is the floating point type among the data types of the inputs with the most precision.

- Otherwise, if the data type of one of the inputs is a fixed-point type, the data type of the output is the smallest fixed-point type capable of representing the result without any loss of precision.
- Otherwise, if the inputs are integral, the output is integral. If any input is signed, the output is signed. The bit width of the output is the largest among the bit widths of the inputs.

Parameters

- **Number of inputs:**

  This parameter determines the number of inputs.

Table 93: **Number of Inputs**

Choices	Description
1	Initially, the block icon is created with a single input.
N	A positive integer value. The block icon is redrawn with the specified number of input ports.

Model Composer Hub

Control implementation of the model.

Description

The Model Composer Hub block controls the behavior of the Vitis Model Composer tool.

You can specify the targeted design flow for the generated output, the directory path for the output, and the desired device and design clock frequency using the following tabs.

- The **Code Generation** tab provides options to select the output flow through Subsystem name, Code directory, Target, and Create testbench options.
- The **Hardware** tab helps with device or board selection. You can specify clock frequency and throughput factor for the model.
- The **Feedback** tab is used to provide feedback to the tool developers, and suggestions to improve the tool.

Library

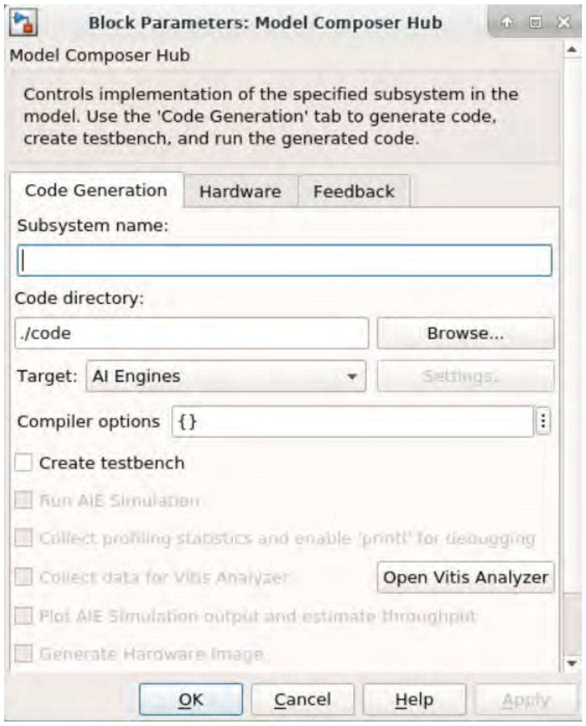
Tools

Data Type Support

Data type support is not applicable to the Model Composer Hub block.

Parameters

Figure 405: **Model Composer Block Parameters**



Block Parameters: Model Composer Hub

Model Composer Hub

Controls implementation of the specified subsystem in the model. Use the 'Code Generation' tab to generate code, create testbench, and run the generated code.

Subsystem name:

Code directory: ./code

Target: AI Engines

Compiler options {}

Create testbench

Run AIE Simulation

Collect profiling statistics and enable 'printf' for debugging

Collect data for Vitis Analyzer

Plot AIE Simulation output and estimate throughput

Generate Hardware Image

- **Code Generation tab:**
  - **Subsystem name:** Enter a subsystem name that contains only Model Composer blocks.
  - **Code directory:** Enter the output directory name with the complete path, or use the **Browse** button to provide a path.
  - **Target:**

    Target settings are shown in the following table.

Table 94: **Target Settings**

Setting	Description
IP Catalog	Select **IP Catalog** to export the design to the Vivado IP Catalog. After C/C++ code generation, Vitis High-Level Synthesis (HLS) is invoked to synthesize the C code and create a project that can be exported as an IP to the Vivado IP Catalog.

Table 94: **Target Settings** *(cont'd)*

Setting	Description
System Generator	Select **System Generator** to export the design to HDL blockset. After C/C++ code generation, Vitis High-Level Synthesis (HLS) is invoked to synthesize the C code and create an RTL solution that can be used as a Vitis HLS block in a HDL model.
HLS C++ code	Select **HLS C++ code** to compile the design model into C++ code.
AI Engines	This is the default selection which allows you to generate the dataflow graph code and verify it using the AIE Simulation.

- **Compiler options:** When enabled, this option provides control over compiler debug options, execution target options etc.

  ***Note*:** This option is only available when the target is AI Engines.

- **Create testbench:** When enabled, Model Composer generates the test vectors while generating the code.

- **Run AIE Simulation:** This option is only available if **Create testbench** is selected. When enabled, it runs the AIE simulation after code generation.

  ***Note*:** This option is only available when the target is AI Engines.

- **Collect profiling statistics and enable 'printf' for debugging:** When enabled, this option allows profiling data to be collected for analysis.

  ***Note*:** This option is only available when the target is AI Engines.

- **Collect data for Vitis Analyzer:** When enabled, this option provides a summary of the simulation results which can be viewed in the Vitis Analyzer.

  ***Note*:** This option is only available when the target is AI Engines.

- **Open Vitis Analyzer:** Click to invoke the Vitis Analyzer tool.

  ***Note*:** This option is only enabled after AIE Simulation has been ran at least once after enabling the **Collect Data for Vitis Analyzer** option.

- **Plot AIE Simulation output and estimate throughput:** When enabled, this option logs simulation data and allows visualization of the output of an AI Engine subsystem.

  ***Note*:** This option is only available when the target is AI Engines.

- **Generate Hardware Image:** When this option is enabled, the tool generates a boot image which can be programmed on a Versal device.

  ***Note*:** This option is only available to select when a valid platform is specified in the Hardware tab.

- **Create and run testbench:**

If selected, Model Composer runs simulation and generates test vectors while generating code.

> **IMPORTANT!** *This option is only available when the target is set to IP Catalog, System Generator, or HLS C++ code.*

- **Testbench stack size:**

  This parameter prompts you to enter a larger stack size.

  When **Create and run testbench** is enabled, the **Testbench stack size** option specifies the size of the testbench stack frame during C simulation (CSIM). Occasionally, the default stack frame size of 10 MB allocated for execution of the testbench may be insufficient to run the test, due to large arrays allocated on the stack and/or deep nesting of sub-systems. Typically when this happens, the test would fail with a segmentation fault and an associated error message. In such a case you may opt to increase the size of the stack frame and rerun the test.

- **Hardware tab:**

  - **Platform:** By default no platform is specified. Selecting **Specify Platform** from the dropdown, allows you to specify a valid hardware platform by browsing to a particular location. It is not mandatory to specify a platform for generating the data-flow graph code.

  The following options are available only when the target is set to IP Catalog, System Generator, or HLS C++ code:

  - **Project device:**

    Specifies the current target part or board platform for the Model Composer model.

    Clicking the browse button (**...**) next to **Project device** displays the Device Chooser dialog box, which allows you to select the board or part to which your design is targeted. Vitis Model Composer obtains board and device data from the Vivado database.

  - **FPGA clock frequency:**

    Specifies the clock frequency in MHz for the Xilinx device. This frequency is passed to the downstream tool flow.

  - **Throughput factor:**

    Specifies the number of samples processed per clock to increase the throughput. A larger factor increases hardware resource usage. The throughput factor must be between 1 and 16.

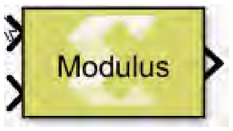
- **Feedback tab:** This tab requests feedback to the tool developers, and suggestions to improve the tool. It points to a weblink that opens a survey that can be completed in less than 3 minutes.

Modulus

Performs element-wise modulus operation on the input signals

Library

Math Functions / Math Operations



Description

The Modulus block takes two inputs. The first input is taken as dividend and the second input is considered as divisor. The output is the remainder after division. For each element of the dividend A, compute the modulus operation (remainder after division) with regard to the corresponding element of the divisor B, as follows:

```
M = mod(A, B) = A - B .* floor(A ./ B)
```

The block can handle division by 0 by returning the NaN output for floating-point data types and 0 for the rest of the data types.

Note: For signed data types the remainder will have the same sign as the divisor B. If instead it should have the same sign as the dividend, use the Remainder block instead of the Modulus block.

Data Type Support

Data types accepted at the inputs of the block are:

- The block supports all native Simulink® data types, as well as half precision floating-point, and fixed-point data types.
- For inputs of bool data type, the output value is always false.
- The block supports scalars, vectors, and 2D matrices.
- The input dimensions must match unless one input is a scalar, in which case, it gets broadcast to be used with each of the other input's elements. The output dimension is the larger of the two input dimensions.
- The block supports mismatched input data types. The output data type is decided by Vitis Model Composer data type propagation rules.
- The block operates on inputs of real numeric type only. For input of complex numeric type it issues an error.

Parameters

The Modulus block has no parameters to set.

Mux

Combines scalar and vector inputs into a larger vector output.

Library

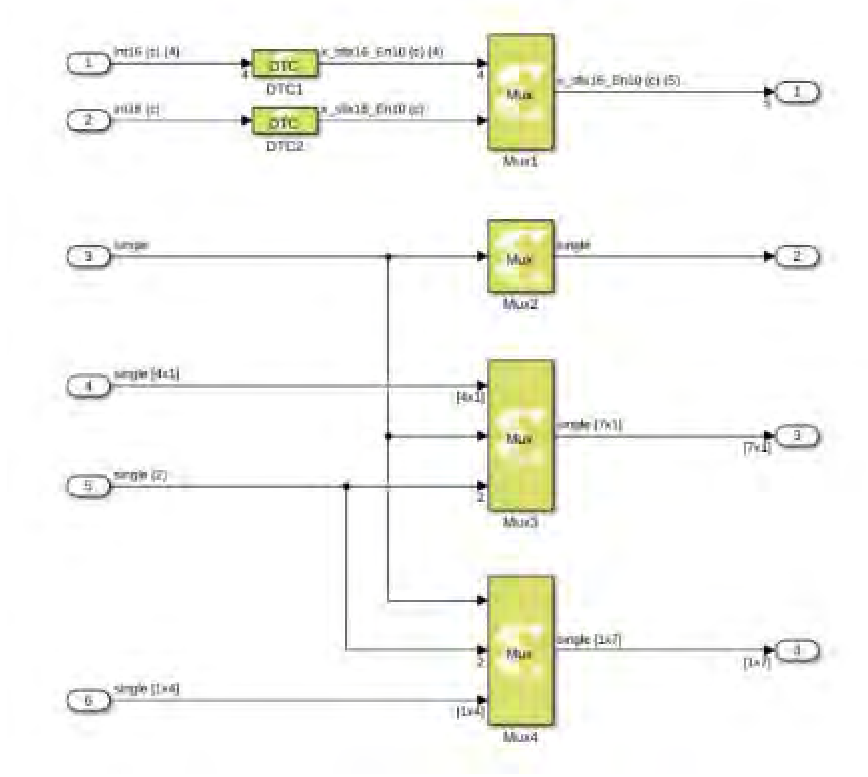
Signal Routing

Description

The Mux block combines scalar and vector inputs into a larger vector. The elements of the inputs are concatenated starting from the first input at the top left. The block takes an input signal that is a scalar, a vector, a row matrix, or a column matrix with the limitation that it cannot support a row matrix signal, and a column matrix signal at the same time. If an input is a row vector or a column vector, then the output also takes that form. The output is a non-virtual vector meaning that its elements are stored in contiguous memory.

The number of inputs to the block is configurable using the Number of inputs block parameter. When the value of the block parameter is changed, the input ports are added or removed starting from the last port at the bottom left.

Figure 406: **Mux Diagram**



Note: This figure shows how the Mux block computes output port dimensions.

Data Type Support

- **Inputs:**
  - The number of inputs is decided by value of the Number of inputs parameter.
  - The input signal can be a scalar, a vector (M), a row matrix (1xM), or column matrix (Mx1).
  - The block cannot have a row matrix and a column matrix as inputs at the same time.
  - All inputs must have the same data type, and the same numeric type either real or complex.
  - The Mux block supports all native data types (double, single, uint8, int8, uint16, int16, uint32, int32, and boolean), and Model Composer supported half and fixed-point data types.
- **Outputs:**
  - The block has one output port.
  - The output data type and numeric type are the same as the inputs.
  - The output signal dimension depends upon the dimensions of the input signals.

Parameters

- **Number of inputs:**

  The value for the parameter must be a finite positive integer. When the value of the parameter changes, the input ports are either added or removed starting from the last port at the bottom left.

Table 95: **Number of Inputs Settings**

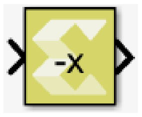
Choices	Description
2	The block icon is initially created with two input ports.
N	A finite positive integer value. The block icon is redrawn with the specified number of input ports.

Negate

Perform element-wise unary minus operation on the input data

Library

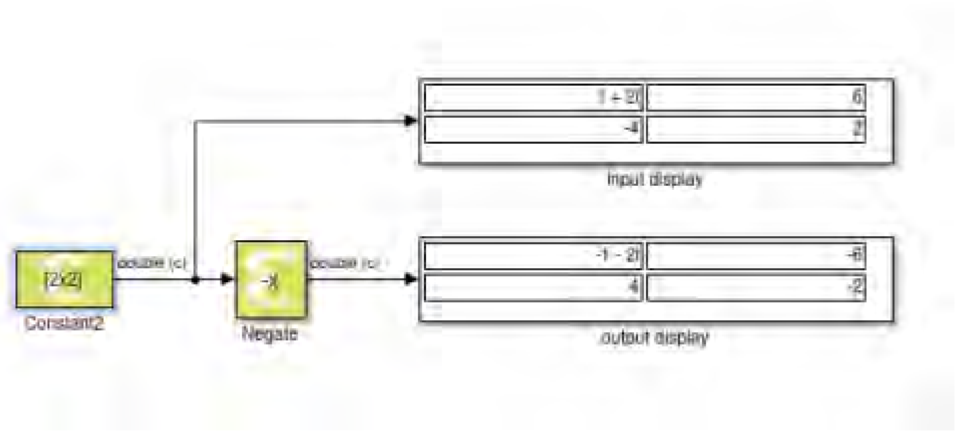
Math Functions / Math Operations



Description

The Negate block computes element-wise minus operation on the input data. The block handles signedness of real and imaginary parts separately in case of complex data input.

Figure 407: **Negate Block**



Data Type Support

Data type support is:

- Input Data Type: All data types are supported except unsigned integer and Boolean values.
- Output: The data type, dimension, and complexity of the output are the same as those of the input signal.

The block supports scalar, vector, and two-dimensional matrix data.

Parameters

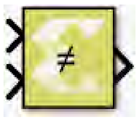
The Negate block has no parameters to set.

Not Equals

Perform element-wise not equal to relational operation on the inputs. The top input corresponds to the first operand.

Library

Relational Operations



Description

The Not Equals block has two input signals and one output signal. The block compares two inputs using element-wise not equals relational operation. The first input corresponds to the top input port and the second input to the bottom input port. The dimension of the output signal matches the dimensions of the input signals. An element of the output signal is true if the corresponding element of the first input signal is not equal to the corresponding element of the second signal; otherwise the element is false.

Data Type Support

Data type support for the block is:

- The data types of the input signals can be of any integer, fixed-point, boolean, or floating-point data type.
- The input signals can be a scalars, vectors, or matrices. If both inputs are not scalar, their dimension must match.
- The input signals can be complex.

- The output signal is boolean.
- The dimension of the output signal is scalar if both inputs are scalar. Otherwise, it matches the dimensions of the non-scalar input.

Parameters

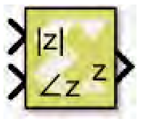
The Not Equals block has no parameters to set.

Polar to Complex

Element-wise conversion of real magnitude and angle representation signals into a complex signal

Library

Math Functions / Math Operations



Description

The Polar to Complex block accepts a real signal of floating point data type such as double, single, or half. The first and second inputs represent magnitude and angle respectively. The angle is in radians. The outputs are complex values of the same data type as the block input for a given real magnitude and angle. The input can be scalar, vector or matrix of real signals, in which case the output signals are also scalar, vector, or matrix. The elements of a certain magnitude input map to the magnitudes of the corresponding complex output elements. Similarly, the elements of a certain angle input map to the angles of the corresponding complex output elements.

Data Type Support

Data types accepted at the inputs of the block are:

- Dimension : The inputs can be scalar, array, or combination of scalar and an array. If both the inputs are arrays, they must have the same dimensions.
- Data Types: Supports signal of floating point data type such as double, single, and half. Both inputs must have the same data type.
- Complex Number Support: No

Outputs for the different input types are:

Table 96: **Input/Output**

Inputs	Output
Both are scalar	Scalar
Both are vector	Vector of same dimension
Both are matrix	Matrix of same dimension
One is scalar and the other is vector or matrix	Dimension is that of the vector or matrix

Parameters

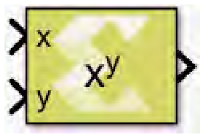
The Polar to Complex block has no parameters to set.

Pow

Compute the element-wise power function

Library

Math Functions / Math Operations



Description

The Pow block computes the value of a base raised to the power exponent. Z = X^y raises each element of X to the corresponding power in y. If one of inputs is a matrix and the other is a scalar, the scalar input is expanded to match the dimension of the non-scalar input to perform the operation. If both inputs are non-scalar, they must agree in dimension.

Data Type Support

Data type support for the block is:

- Dimension: Inputs can be scalar, vector, or matrix. If one of the inputs is scalar and the other is a vector or matrix then the scalar input is expanded to match the other input dimension and the operation will be performed element-wise. If both inputs are non-scalar, then they must match in dimension.
- Data Types: Input supports signals of integer type, floating point data type (double, single, and half), and signed and unsigned fixed-point type. Both inputs must be of the same data type.
- Complex Numbers: Complex numbers are not supported.

A negative base value raised to a fractional power will result in a Not A Number (NAN) value.

Parameters

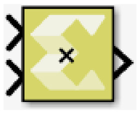
The Pow block has no parameters to set.

Product

Compute element-wise product of the input signals

Library

Math Functions / Math Operations



Description

The Product block computes the element-wise product of its input signals.

The block warns or errors out when an integer output overflows during simulation. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink® Editor, then set the **Wrap on overflow**, or **Saturate on overflow** parameter.

Data Type Support

The Product block outputs the result of multiplying two inputs. Inputs can be two scalars, a scalar and a non-scalar, and two non-scalars of the same dimension. If one of the inputs is scalar and other input is vector/matrix then the scalar input is expanded to match the dimension of the other input to perform the operation. If input signals are both integer data types, such as int16, the datatype of the output is also an integer datatype int16. Hence an overflow of wrap around (no saturation) is likely if the output exceeds the range that is represented with int16. If the operation is to be applied without loss of precision or range, use fixed-point data types. If you want to narrow the bit width of the output signal, you can run it through a Data Type Conversion block and select a fixed-point data type that saturates.

The output dimension is the same as that of the inputs if both inputs are either scalar or non scalar. If one input is scalar and the other is a vector/matrix then the output dimension matches the dimension of the vector/matrix input. The output type is same as that of the inputs if both inputs are of the same type. Otherwise, the output type is defined as follows.

Table 97: **Input/Output**

Input Type	Output Type
(double, single)	double

Table 97: **Input/Output** *(cont'd)*

Input Type	Output Type
(double, int)	double
(double, half)	double
(double, fixed)	double
(double, Boolean)	double
(single, int)	single
(single, half)	single
(single, fixed)	single
(single, Boolean)	single
(half, int)	half
(half, fixed)	half
(half, Boolean)	half
(fixed, int)	fixed
(fixed, Boolean)	fixed
(int, Boolean)	int

Parameters

Saturate on integer overflow

This parameter specifies whether integer overflow is handled by wrapping (default) or by saturating. This parameter is relevant only if the output is integral (int8, int16, int32, uint8, uint16, uint32).

Settings for the **Saturate on integer overflow** parameter are:

Table 98: **Saturate on Integer Overflow Parameter**

Setting	Description
Not selected	Integer overflow is handled by wrapping.
Selected	Integer overflow is handled by saturation.

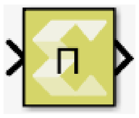
When overflow is detected, the Diagnostic Viewer displays messages that depend on the diagnostic action you specify in the Simulink Editor. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink Editor, then set the **Wrap on overflow** or **Saturate on overflow** parameter.

Product of Elements

Multiply the elements of the input signal

Library

Math Functions / Math Operations



Description

The Product of Elements block computes the product of the elements of the input signal. The block can be configured in the following ways.

- By default, the output is scalar and equal to the product of all (matrix) elements of the input signal.
- If the dimension to multiply over is specified to be **1**, the output is a row matrix (1xN), where N is the number of columns of the input, and element (1,k) is the product of the elements of column k of the input.
- If the dimension to multiply over is specified to be **2**, the output is a column matrix (Mx1), where M is the number of rows of the input, and element (k,1) is the product of the elements of row k of the input.

Data Type Support

The input signal can be real or complex. The input data type can be any Boolean, integer, floating-point, or fixed-point data type. The block can perform element-wise multiplication on real or complex number data.

Parameter

Multiply over

The **Multiply over** parameter value is used to decide whether elements will be multiplied in all dimensions or in one of the dimensions.

Following are the settings for the **Multiply over** parameter.

Table 99: **Multiply Over Parameter**

Setting	Description
All dimensions	Multiply all elements of the input signal (output is scalar)
Specified dimension	This option shows an edit box, **Dimension**, where the specific dimension value can be entered.

Dimension

The **Dimension** parameter is displayed only if the **Multiply over** parameter value is set to **Specified dimension**.

Following are the settings for the **Dimension** parameter.

Table 100: **Dimension Parameter**

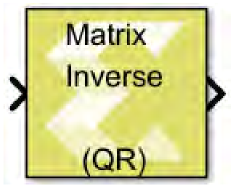
Setting	Description
1	Multiply input over row dimension. Output is a row matrix.
2	Multiply input over column dimension. Output is a column matrix.

QR Inverse

Compute the inverse of a matrix using QR factorization

Library

Math Functions / Matrices and Linear Algebra



Description

The QR Inverse block provides the inverse of the input matrix A by performing QR factorization.

$$A^{-1} = Q^{T}R^{-1}$$

Q is an orthogonal matrix and R is an upper triangular square matrix. For singular matrix input the output would contain NaN or +inf/-inf.

Data Type Support

Data type support is:

- Dimension: Input has to be a square matrix. Scalar and vector inputs are not supported.
- Data Types: Input supports signals of floating point data types (double, single, and half). It does not support integer, boolean, and fixed-point data types.
- Complex Numbers: Complex numbers are not supported.

The output has the same dimension and data type as the input.

Parameters

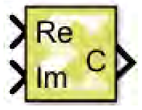
The QR Inverse block has no parameters to set.

Real-Imag to Complex

Computes the complex output from real and imaginary input.

Library

Math Functions / Math Operations



Description

The Real-Imag to Complex block converts the real and imaginary inputs to a complex-valued output signal. The input signal can be of any data type except boolean.. The complex output has the same data type as that of the block input. The input can be a scalar, 1-D vector, or matrix of real signals. It is possible to specify the constant real or imaginary part from the block dialog.

Data Type Support

Data type support for the input port is as follows.

- The block supports all data types except boolean. If the 'Input' is 'Real and Imag', both the inputs must have the same data type. Otherwise, 'Real part' or 'Imag part' parameter is converted to the same data type as that of the block input.
- Real and imaginary parts specified using inputs or block dialog must be real.
- The input can be a scalar, 1-D vector, or matrix.
- The output is always complex, and has the same data type as that of the input.
- Dimensions:
  - The output has the same dimensions as that of the input when both real and imaginary parts have the same dimensions.
  - The block supports mixed dimensions for real and imaginary inputs (specified as inputs or using block dialog). Any scalar input is expanded to the dimensions of non-scalar input, and the block has the same dimensions as those of the non-scalar input. If both the inputs are non-scalar, they must agree on dimensions.

Parameters

- **Input:**

Input is a drop down menu parameter which specifies whether real, imaginary, or both of the parts of the output signal are specified as inputs.

- **Settings:**

  Following are settings for the **Input** parameter.

Table 101: **Input Parameter**

Setting	Description
Real and imag	Real and imaginary parts of the output signal are specified using Re and Im inputs of the block, respectively.
Real	The block has only Re input in this case. Real part of the output signal is specified using the Re input of the block, while its imaginary part is specified using the Imag part parameter.
Imag	The block has only Im input in this case. Imaginary part of the output signal is specified using the Im input of the block, while its real part is specified using the Real part parameter.

- **Real part:** Specify the constant real part of the output signal when Input is set to Imag. This parameter is visible only when you set Input to Imag.

Table 102: **Real Part Parameter**

Choices	Description
0	The value of the Real part parameter must be a numeric, real-valued scalar, vector, or matrix.

- **Imag part:** Specify the constant imaginary part of the output signal when Input is set to Real. This parameter is visible only when you set Input to Real.

Table 103: **Imag Part Parameter**

Choices	Description
0	The value of the Imag part parameter must be a numeric, real-valued scalar, vector, or matrix.

Reciprocal

Element-wise computation of the reciprocal for a given argument

Library

Math Functions / Math Operations

Description

The Reciprocal block returns the output of the function inv(`x`) for each element in array `x`. The block supports input of all data types except Boolean. The input can be scalar, vector or matrix.

Data Type Support

Data types accepted at the inputs of the block:

- Dimension: Input can be scalar, vector, or matrix.
- Data Types: Input supports signals of integer, fixed-point, and floating-point data type. The block does not support Boolean inputs.
- Complex Number Support: No

  The output has the same dimension and type as the input.

Parameters

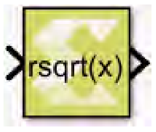
The Reciprocal block has no parameters to set.

Reciprocal Sqrt

Element-wise computation of the reciprocal square root for a given argument

Library

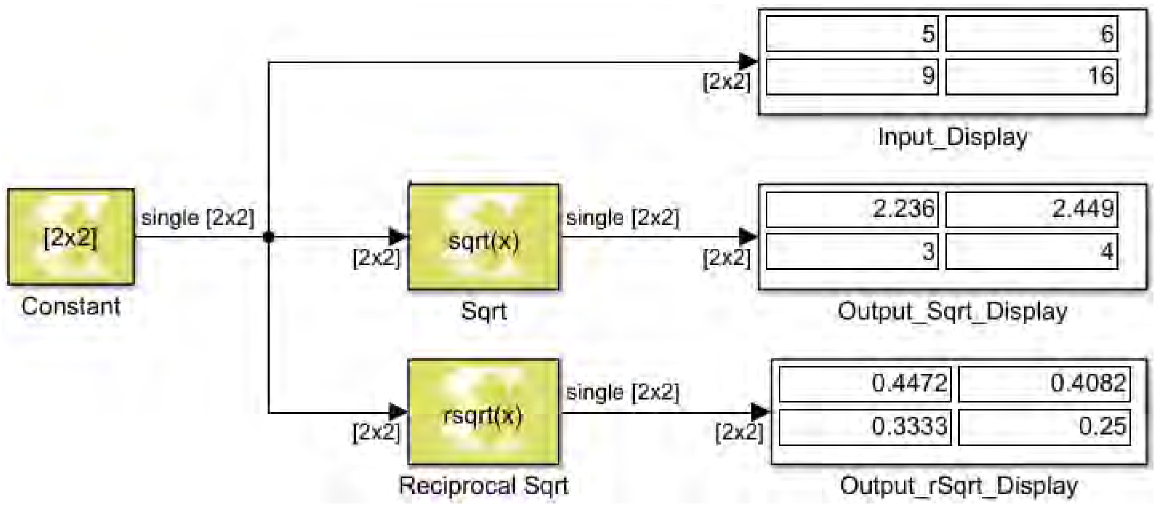
Math Functions / Math Operations



Description

The Reciprocal Sqrt block returns the reciprocal square root for each element in an array. The block supports input of all data types except Boolean. The input can be a scalar, vector or a matrix.

Figure 408: **Reciprocal Sqrt Block**



Data Type Support

Data type support is:

- Dimension: Input can be scalar, vector or matrix.
- Data Types: Input supports signals of integer, fixed-point, and floating point data type. It does not support Boolean inputs.
- Complex Numbers: Complex numbers are not supported.

The output has the same dimension and data type as the input.

Parameters

The Reciprocal Square root block has no parameters to set.

Reduction AND

Compute bitwise AND of the elements of the input over all dimensions or over a specified dimension

Library

Logic and Bit Operations

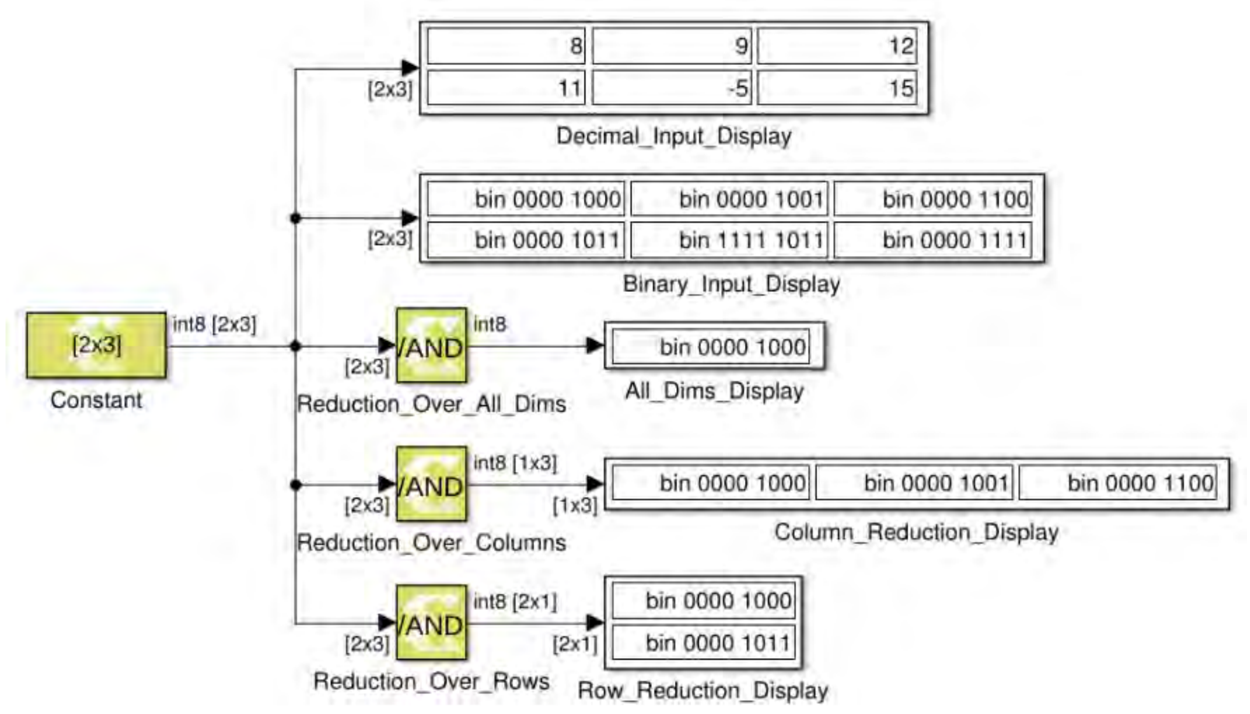
Description

The Reduction AND block has one input signal and one output signal. It computes the bitwise AND of the elements of the input signal over all of the dimensions or over a specified dimension. The data type of the output signal is the same as that of the input signal. The dimension of the output signals depends on whether the reduction takes place over all dimensions or over a specified dimension.

- **Reduce over all dimensions:** The output is a scalar and it is the bitwise AND of the elements of the input signal.
- **Reduce over dimension 1:** The output is a row vector (2-D) with as many elements as the number of columns of the input. Each element in the output is the bitwise AND reduction of the elements of the corresponding column of the input.
- **Reduce over dimension 2:** The output is a column vector (2-D) containing as many elements as the number of rows of the input. Each element in the output is the bitwise AND reduction of the elements of the corresponding row of the input.

In the example below a 2x3 input signal of type int8 feeds into three different configurations of the Reduction AND block.

Figure 409: **Reduction AND Block**



Data Type Support

- The input signal can be of any data type except for floating point types.
- The input signal must be real.
- The input signal must be a matrix if reduction is along dimension 2.
- The input signal can be a matrix, vector or scalar if reduction is along all dimensions or along dimension 1.

Parameters

Reduce over

This parameter specifies whether reduction takes place over all dimensions or over a specified dimension. If reduction is specified over all dimensions, the output signal is a scalar.

Following are the settings for the **Reduce over** parameter.

Setting	Description
All dimensions	The Reduce AND operator will be applied to all elements, producing a scalar output.
Specified dimension	The Reduce AND operator will be applied along the specified dimension, producing a vector output along the opposite dimension. When **Dimension** 1 is specified, the Reduction AND is applied along columns, producing a row vector as output. When **Dimension** 2 is specified, the Reduction AND is applied along rows, producing a column vector as output.

Note: The dimension specified will be the one that gets reduced to size 1. For example, a 2-D M x N input matrix specifying **Dimension** 1 (number of rows M) will result in a 1 x N row vector.

Dimension

If the **Reduce over** parameter is set to **Specified dimension**, the **Dimension** parameter specifies over which dimension reduction takes place.

- If the input signal has dimensions M x N and the reduction **Dimension** is 1, the output has dimensions 1 x N.
- If the input signal has dimensions M x N and the reduction **Dimension** is 2, the output has dimensions M x 1.
- If the input signal is scalar or 1 x 1, the output dimension is 1 x 1.

Following are the settings for the **Dimension** parameter.

Table 104: **Dimension Parameter**

Setting	Description
1	Reduce over row dimension.
2	Reduce over column dimension.

Note: If the reduce **Dimension** is specified to be **2** the input signal must be two-dimensional.

Reduction OR

Compute bitwise OR of the elements of the input over all dimensions or over a specified dimension

Library

Logic and Bit Operations

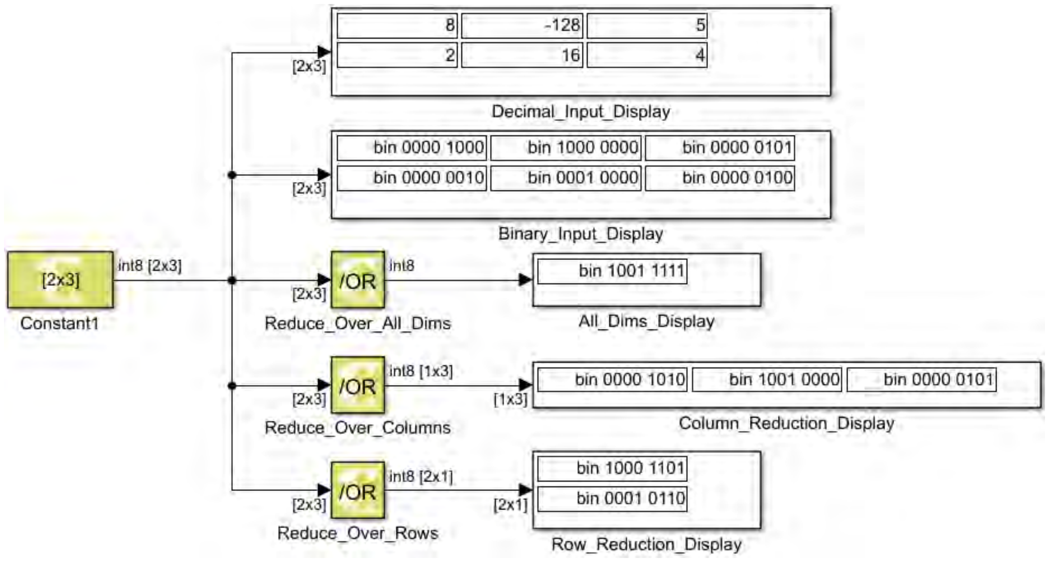
Description

The Reduction OR block has one input signal and one output signal. It computes the bitwise OR of the elements of the input signal over all of the dimensions or over a specified dimension. The data type of the output signal is the same as that of the input signal. The dimension of the output signals depends on whether the reduction takes place over all dimensions or over a specified dimension.

- **Reduce over all dimensions:** The output is a scalar and it is the bitwise OR of all of the elements of the input signal.
- **Reduce over dimension 1:** The output is a row vector (2-D) with as many elements as the number of columns of the input. Each element in the output is the bitwise OR reduction of the elements of the corresponding column of the input.
- **Reduce over dimension 2:** The output is a column vector (2-D) containing as many elements as the number of rows of the input. Each element in the output is the bitwise OR reduction of the elements of the corresponding row of the input.

In the example below a 2x3 input signal of type int8 feeds into three different configurations of the Reduction OR block.

Figure 410: **Reduction OR Block**



Data Type Support

- The input signal can be of any data type except for floating point types.
- The input signal must be real.
- The input signal must be a matrix if reduction is along dimension 2.
- The input signal can be a matrix, vector, or scalar if reduction is along all dimensions or along dimension 1.

Parameters

Reduce over

This parameter specifies whether reduction takes place over all dimensions or over a specified dimension. If reduction is specified over all dimensions, the output signal is a scalar.

Following are settings for the **Reduce over** parameter.

Table 105: **Reduce Over Parameter**

Setting	Description
All dimensions	Reduction takes place over all dimensions.
Specified dimension	Reduction takes place over the dimension specified by the **Dimension** parameter.

Dimension

If the **Reduce over** parameter is set to **Specified dimension**, the **Dimension** parameter specifies over which dimension reduction takes place.

- If the input signal has dimensions M x N and the reduction **Dimension** is 1, the output has dimensions 1 x N.
- If the input signal has dimensions M x N and the reduction **Dimension** is 2, the output has dimensions M x 1.
- If the input signal is scalar or 1 x 1, the output dimension is 1 x 1.

Following are settings for the **Dimension** parameter.

Table 106: **Dimension Parameter**

Setting	Description
1	Reduce over row dimension.
2	Reduce over column dimension.

Note: If the reduce **Dimension** is specified to be **2** the input signal must be two-dimensional.

Reduction XOR

Compute bitwise XOR of the elements of the input over all dimensions or over a specified dimension

Library

Logic and Bit Operations

Description

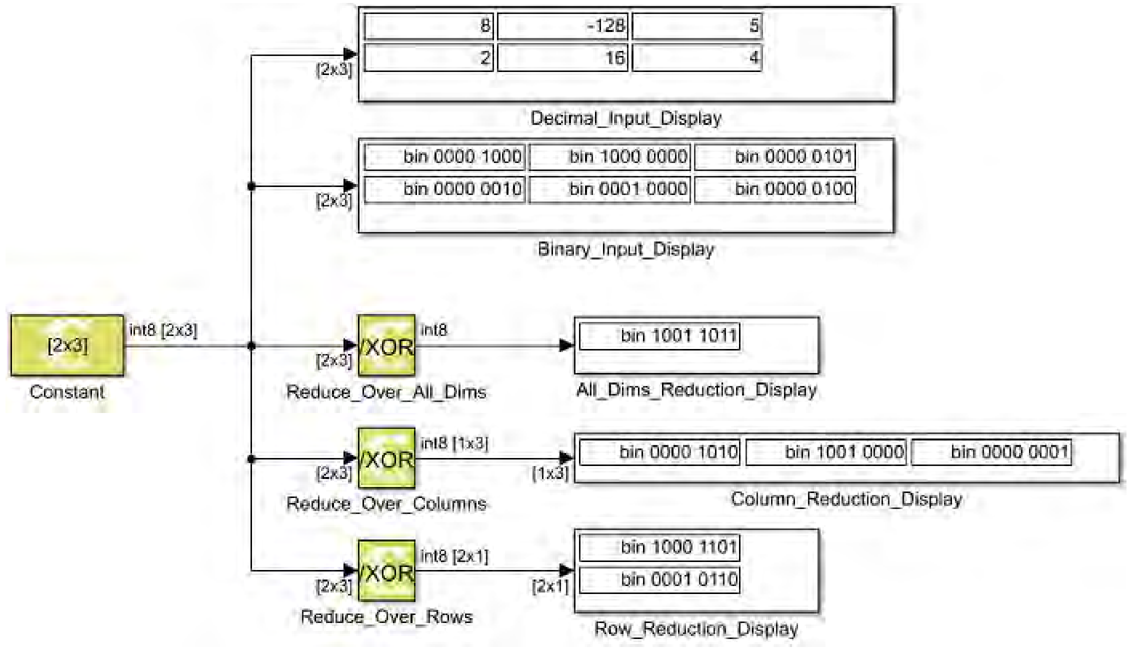
The Reduction XOR block has one input signal and one output signal. It computes the bitwise exclusive OR (XOR) of the elements of the input signal over all of the dimensions or over a specified dimension. The data type of the output signal is the same as that of the input signal. The dimension of the output signals depends on whether the reduction takes place over all dimensions or over a specified dimension.

- **Reduce over all dimensions:** The output is a scalar and it is the bitwise XOR of all of the elements of the input signal.
- **Reduce over dimension 1:** The output is a row vector (2-D) with as many elements as the number of columns of the input. Each element in the output is the bitwise XOR reduction of the elements of the corresponding column of the input.

- **Reduce over dimension 2:** The output is a column vector (2-D) containing as many elements as the number of rows of the input. Each element in the output is the bitwise XOR reduction of the elements of the corresponding row of the input.

In the example below a 2x3 input signal of type int8 feeds into three different configurations of the Reduction XOR block.

Figure 411: **Reduction XOR Block**



Data Type Support

- The input signal can be of any data type except for floating point types.
- The input signal must be real.
- The input signal must be a matrix if reduction is along dimension 2.
- The input signal can be a matrix, vector or scalar if reduction is along all dimensions or along dimension 1.

Parameters

Reduce over

This parameter specifies whether reduction takes place over all dimensions or over a specified dimension. If reduction is specified over all dimensions, the output signal is a scalar.

Settings for the **Reduce over** parameter are:

Table 107: **Reduce Over Parameter**

Setting	Description
All dimensions	Reduction takes place over all dimensions.
Specified dimension	Reduction takes place over the dimension specified by the **Dimension** parameter.

Dimension

If the **Reduce over** parameter is set to **Specified dimension**, the **Dimension** parameter specifies over which dimension reduction takes place.

- If the input signal has dimensions M x N and the reduction **Dimension** is 1, the output has dimensions 1 x N.
- If the input signal has dimensions M x N and the reduction **Dimension** is 2, the output has dimensions M x 1.
- If the input signal is scalar or 1 x 1, the output dimension is 1 x 1.

Settings for the **Dimension** parameter are:

Table 108: **Dimension Parameter**

Setting	Description
1	Reduce over row dimension.
2	Reduce over column dimension.

***Note*:** If the reduce **Dimension** is specified to be **2** the input signal must be two-dimensional.

Reinterpret

Element-wise reinterpretation of the input type into a compatible output type with the same bit width

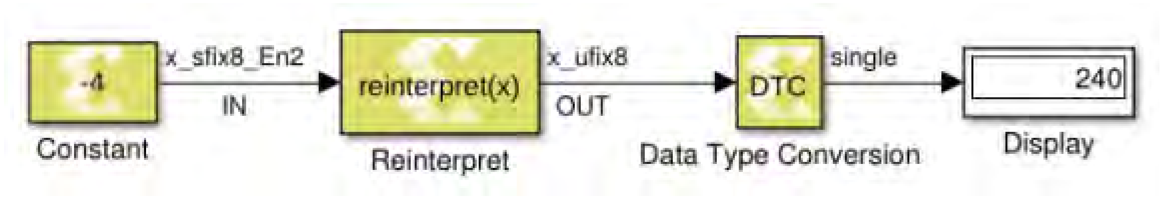
Library

Signal Attributes

Description

The Reinterpret block provides a mechanism for interpreting a value from a different data type. You can specify the output data type with the restriction that the bit widths of input and output data types must match.

Figure 412: **Reinterpret Block**



In the above example, the input is of fixed-point signed number (x_sfix8_En2) represented with 8 bits, in which 2 bits are used for the fractional part (i.e., -4 = 1111 00.00). The Reinterpret block interprets the input type to unsigned fixed-point number (x_ufix8), represented with 8 bits as that of input, and no bits are used for the fractional part. The output becomes 240 (1111 0000).

Data Type Support

The data types of the input can be any integer, Boolean, fixed-point, or floating-point data type. The input can be any real or complex valued signal. If the input is real, the output is real. If the input is complex, the output is complex. The block supports scalar, vector, or matrix data.

Following are the supported output data types.

Table 109: **Input/Output**

Input Data Type	Supported Output Data Type
double	double, 64 bit fixed-point data
single	single, int32, uint32, 32 bit fixed-point data
int8	int8, uint8, 8 bit fixed-point data
uint8	uint8, int8, 8 bit fixed-point data
int16	int16, uint16, half, 16 bit fixed-point data
int16	uint16, int16, half, 16 bit fixed-point data
uint16	int32, uint32, single, 32 bit fixed-point data
int32	int32, uint32, single, 32 bit fixed-point data
uint32	uint32, int32, single, 32 bit fixed-point data
bool	bool, 1 bit fixed-point data
fixed	Same bit width fixed-point data with different fractional widths, all native data types if the bit width matches
half	half, int16, uint16, 16 bit fixed-point data type

Parameters

Output data type

This parameter specifies the output data type for reinterpreting the input data. If **fixed** is specified more parameters are available.

Following are the settings for the **Output data type** parameter.

Table 110: **Output Data Type Parameter**

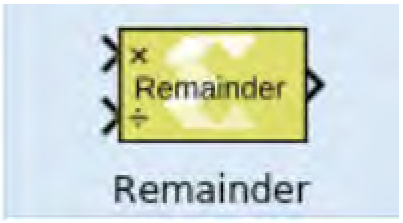
Setting	Description
double	Double precision floating point
single	Single precision floating point
int8	8-bit signed integer
uint8	8-bit unsigned integer
int16	16-bit signed integer
uint16	16-bit unsigned integer
int32	32-bit signed integer
uint32	32-bit unsigned integer
boolean	Boolean
fixed	Fixed-point
half	Half precision floating-point
data type expression	A string that specifies the output data type. See Working with Data Type Expression in the *Vitis Model Composer User Guide* (UG1483).

Remainder

Perform element-wise division on the input signal. The output is the remainder after the division.

Library

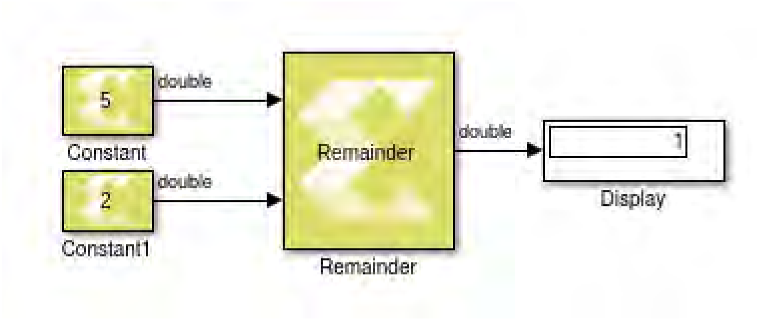
Math Functions / Math Operations



Description

The remainder block takes two inputs. The dividend is the top input and the divisor is the bottom input. The block supports scalar, vector and matrix dimensions. The dimensions of the inputs must match unless one input is a scalar. The output has the larger dimension of the two inputs. The block can handle division by 0 and produces NaN as the output only for floating-point data types double and single.

Figure 413: **Remainder Block**



Data Type Support

The Remainder block supports all MATLAB native data types, half precision floating-point data type, and fixed-point. The block operates on real type inputs where it requires both inputs to have the same type of data.

Parameters

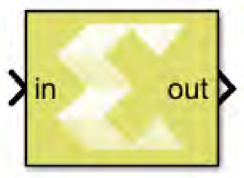
The Remainder block has no parameters to set.

Reshape Row-Major

Changes the input dimensions in row-major order.

Library

Math Functions/Math Operations



Description

The Reshape Row-Major block changes the input dimensions based on the specified Output dimensionality parameter. The output contains elements of the input in row-major order, that is the first row of the input matrix followed by the second row, and so on.

Data Type Support

- The block supports floating point, integer, boolean, and fixed-point data types.
- The block supports real and complex valued inputs.
- The input can be a scalar, 1-D vector, or matrix.
- The output has the same data type as the input.

Parameters

- **Output dimensionality:**

  This parameter specifies how the input should be reshaped. The settings for **Output dimensionality** are as follows:

Table 111: **Output Dimensionality Parameter Settings**

Choices	Description
1-D array	Converts the input to a 1-D vector. For a matrix input, the output consists of input matrix elements in row-major order.
Column vector	Converts the input to a M x 1 matrix, where M is the number of elements in the input signal. For a matrix input, the output consists of input matrix elements in row-major order.
Row vector	Converts the input to a 1 x N matrix, where N is the number of elements in the input signal. For a matrix input, the output consists of input matrix elements in row-major order.
Custom	Converts the input to an output which has dimensions specified by the user using the **Output dimensions** parameter. The conversion is done in row-major order. The value of the **Output dimensions** parameter must be a two-element vector. For example, a value of `[M N]` outputs an M x N matrix. The number of elements of the input must match the number of elements specified by the **Output dimensions** parameter.
Derive from reference input port	Creates a second input port on the block and derives the dimensions of the output from the dimensions of the second input port. Selecting this option disables the **Output dimensions** parameter. Both the inputs to the block must have the same number of elements.

- **Output dimensions:**

  Specify Output dimensions when **Output dimensionality** is set to **Custom**. The settings for **Output dimensions** are as follows:

Table 112: **Output Dimensions Parameter Settings**

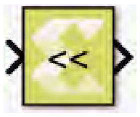
Choices	Description
[1, 1]	The value of the **Output dimensions** parameter must be a two-element vector.

Shift Left

Perform logical shift left of input over a constant number of bit positions specified by a non-negative integer parameter

Library

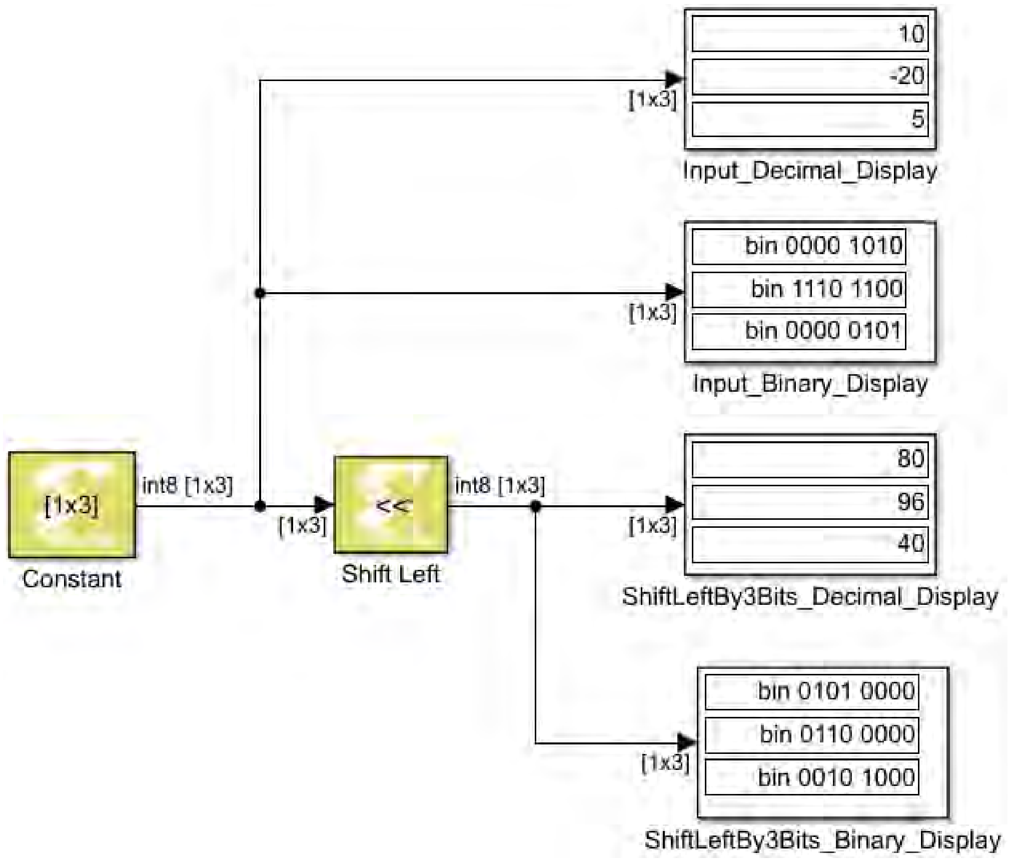
Logic and Bit Operations



Description

The shift left block computes element-wise left shift of the input by a constant amount specified via a parameter. This is also known as a logical shift left. The shift amount specifies over how many bit positions bits are shifted. This shift amount must be a non-negative integer. The default value is 0. The output is of the same type, dimension, and numeric type (real or complex) as the input. The input type must be integral or fixed-point.

Figure 414: **Shift Left Block**



Data Type Support

- Input: integer, fixed-point, but not logical or floating point.
- Output: output data type is the same as input data type.

Parameters

Shift by

This parameter specifies the number of bit positions over which the shift takes place.

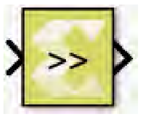
Enter a scalar real non-negative integer for the **Shift by** parameter.

Shift Right

Performs arithmetic shift right of input over a constant number of bit positions specified by a non-negative integer parameter

Library

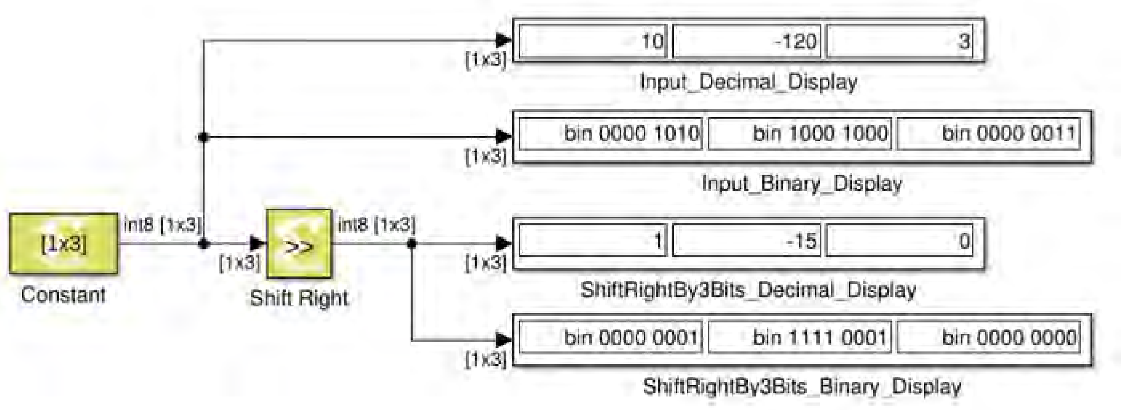
Logic and Bit Operations



Description

The shift right block computes an element-wise right shift of the input by a constant amount specified via a **Shift by** parameter. This is also known as arithmetic shift right. The shift amount specifies over how many bit positions bits are shifted. This shift amount must be a non-negative integer. The default value is 0. The output is of the same type, dimension, and numeric type (real or complex) as the input. The input data must be integer or fixed-point type.

Figure 415: **Shift Right Block**



Data Type Support

The Shift Right block supports integer and fixed-point input data, but not Boolean or floating type. Input data can be real or complex. Output data type and dimension are the same as that of input data.

Parameters

Shift by

This parameter specifies the number of bit positions over which the shift takes place.

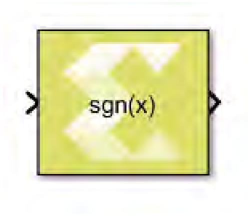
Enter a scalar real non-negative integer for the **Shift by** parameter.

Signum

Performs signum function (sign extraction) on the input.

Library

Math Functions/Math Operations



Description

The Signum block returns the sign for each element of the real input. The block returns 1, 0, or -1 if the number is positive, equal to 0, or negative, respectively.

When the input s is complex, the block output is calculated as:

```
sign(s) = s./ abs(s)
```

Where sign is the MATLAB® signum function, and `./` indicates element-wise division.

Data Type Support

The Signum block supports signals of integer type, floating-point type (double, single and half), and fixed-point type for real inputs. The complex inputs are supported only for floating point data types.

The output data type, and dimension are the same as those of the input.

Parameters

The Signum block has no parameters to set.

Sine

Element-wise computation of the sine function for the given input

Library

Math Functions / Math Operations

Description

The Sine block returns the output of the function sin(`x`) for each element in array `x`.

Data Type Support

Data type support is:

- Dimension: Input can be scalar, vector or matrix.
- Data Types: Input supports signals of floating point data types (double, single, and half) and signed fixed-point type. It does not support integer, Boolean, and unsigned fixed-point data types.
- Complex Numbers: Complex numbers are not supported.

The output has the same dimension and data type as the input. However, If the data type of the input is a fixed-point type, the data type of the output is fixed-point type with integer width fixed as 2.

Parameters

The Sine block has no parameters to set.

sinh

Element-wise computation of the hyperbolic sine for a given argument

Library

Math Functions / Math Operations

Description

The sinh block returns the output of the function sinh(x), which is the hyperbolic sine, for each element in array x.

The hyperbolic sine of `x` is:

$$\sinh(x) = \frac{e^{x} - e^{-x}}{2}$$

Data Type Support

Data type support is:

- Dimension: Input can be scalar, vector or matrix.
- Data Types: Input supports signals of integer type, floating-point data types (double, single, and half) and fixed-point type.
- Complex Numbers: Complex numbers are not supported.

The output has the same dimension and data type as the input.

Parameters

The sinh block has no parameters to set.

Sqrt

Element-wise computation of the square root for a given argument

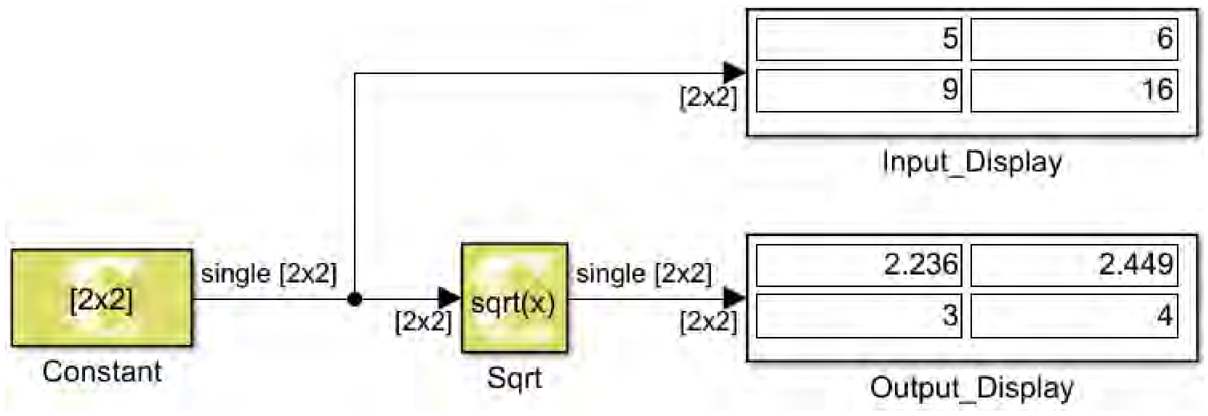
Library

Math Functions / Math Operations

Description

The Sqrt block returns the square root for each element in array x. The block supports input of all data types except boolean. The input can be a scalar, vector or a matrix.

Figure 416: **Sqrt Block**



Data Type Support

Data type support is:

- Dimension: Input can be scalar, vector or matrix.
- Data Types: Input supports signals of integer, fixed-point and floating-point data type. It does not support Boolean inputs.
- Complex Numbers: Complex numbers are not supported.

The output has the same dimension and data type as the input.

Parameters

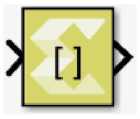
The Sqrt block has no parameters to set.

Submatrix

Select a subset of elements (submatrix) from matrix input

Library

Math Functions / Matrices and Linear Algebra



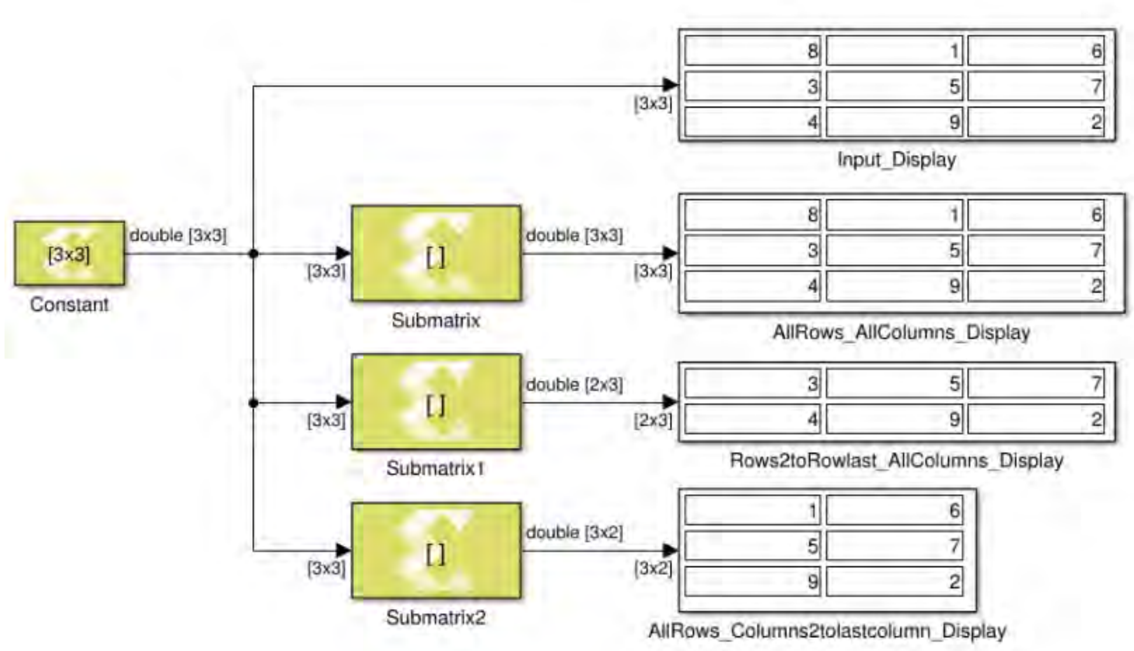
Description

The Submatrix block extracts a contiguous submatrix from the `M`-by-`N` input matrix `u`. The **Row span** parameter provides three options for specifying the range of rows in `u` to be retained in submatrix output `y`:

- **All rows**: Specifies that `y` contains all `M` rows of `u`.
- **One row**: Specifies that `y` contains only one row from `u`. The Row parameter (described below) is enabled to allow selection of the desired row.
- **Range of rows**: Specifies that `y` contains one or more rows from `u`. The **Starting row** and **Ending row parameters** are enabled to allow selection of the desired range of rows.

The Column span parameter contains a corresponding set of three options for specifying the range of columns in `u` to be retained in the submatrix `y`: **All columns**, **One column**, or **Range of columns**. The **One column** option enables the Column parameter, and Range of columns options enable the **Starting column** and **Ending column** parameters.

Figure 417: **Submatrix Block**



Data Type Support

All data types are supported. The output type is the same as the input type.

Parameters

- **Row span:** The range of input rows to be retained in the output. Options are **All rows**, **One row**, or **Range of rows**.
- **Row:** The input row to be used as the row of the output. **Row** is enabled when you select **One row** for **Row span**.
- **Row index:** The index of the input row to be used as the first row of the output. **Row index** is enabled when you select **Index** for **Row**.

- **Row offset:** The offset of the input row to be used as the first row of the output. **Row offset** is enabled when you select **Offset from middle** or **Offset from last** for **Row**.
- **Starting row:** The input row to be used as the first row of the output. **Starting row** is enabled when you select **Range of rows** for **Row span**.
- **Starting row index:** The index of the input row to be used as the first row of the output. **Starting row index** is enabled when you select **Index** for **Starting row**.
- **Starting row offset:** The offset of the input row to be used as the first row of the output. **Starting row offset** is enabled when you select **Offset from middle** or **Offset from last** for **Starting row**.
- **Ending row:** The input row to be used as the last row of the output. **Ending row** is enabled when you select **Range of rows** for **Row span** and you select any option but **Last** for **Starting row**.
- **Ending row index:** The index of the input row to be used as the last row of the output. **Ending row index** is enabled when you select **Index** for **Ending row**.
- **Ending row offset:** The offset of the input row to be used as the last row of the output. **Ending row offset** is enabled when you select **Offset from middle** or **Offset from last** for **Ending row**.
- **Column span:** The range of input columns to be retained in the output. Options are **All columns**, **One column**, or **Range of columns**.
- **Column:** The input column to be used as the column of the output. **Column** is enabled when you select **One column** for **Column span**.
- **Column index:** The index of the input column to be used as the first column of the output. **Column index** is enabled when you select **Index** for **Column**.
- **Column offset:** The offset of the input column to be used as the first column of the output. **Column offset** is enabled when you select **Offset from middle** or **Offset from last** for **Column**.
- **Starting column:** The input column to be used as the first column of the output. **Starting column** is enabled when you select **Range of columns** for **Column span**.
- **Starting column index:** The index of the input column to be used as the first column of the output. **Starting column index** is enabled when you select **Index** for **Starting column**.
- **Starting column offset:** The offset of the input column to be used as the first column of the output. **Starting column offset** is enabled when you select **Offset from middle** or **Offset from last** for **Starting column**.
- **Ending column:** The input column to be used as the last column of the output. **Ending column** is enabled when you select **Range of columns** for **Column span** and you select any option but **Last** for **Starting column**.
- **Ending column index:** The index of the input column to be used as the last column of the output. **Ending column index** is enabled when you select **Index** for **Ending column**.

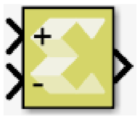
- **Ending column offset:** The offset of the input column to be used as the last column of the output. **Ending column offset** is enabled when you select **Offset from middle** or **Offset from last** for **Ending column**.

Subtract

Perform element-wise subtraction

Library

Math Functions / Math Operations



Description

The Subtract block performs element-wise subtraction of two input signals.

The block warns or errors out when an integer output overflows during simulation. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink Editor, then set the **Wrap on overflow** or **Saturate on overflow** parameter.

Data Type Support

The Subtract block supports any floating point, fixed-point, integer or Boolean data type. The block can perform element-wise subtraction on real and complex inputs. The input signals can be scalars, vectors or matrices. When both inputs have non-scalar dimensions, the dimensions must match each other.

Parameters

Saturate on integer overflow

This parameter specifies whether integer overflow is handled by wrapping (default) or by saturating. This parameter is relevant only if the output is integral (int8, int16, int32, uint8, uint16, uint32).

Settings for the **Saturate on integer overflow** parameter are:

Table 113: **Saturate On Integer Overflow Parameter**

Setting	Description
Not selected	Integer overflow is handled by wrapping.
Selected	Integer overflow is handled by saturation.

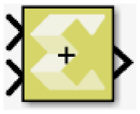
When overflow is detected, the Diagnostic Viewer displays messages that depend on the diagnostic action you specify in the Simulink Editor. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink Editor, then set the **Wrap on overflow** or **Saturate on overflow** parameter.

Sum

Perform element-wise addition of two input signals

Library

Math Functions / Math Operations



Description

The Sum block performs element-wise addition of two input signals.

The block warns or errors out when an integer output overflows during simulation. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink Editor, then set the **Wrap on overflow** or **Saturate on overflow** parameter.

Data Type Support

Data types accepted at the inputs of the block are as follows.

- This block supports all data types supported by Vitis Model Composer. The block supports real and complex numbers.
- The input signals can be real or complex numbers of scalar, vector or matrix type. When both inputs are non-scalar then their dimensions must match.

Output data types are as follows.

- If the data type of one input is a floating point type, the data type of the output is the floating point type among the data types of the inputs with the most precision.
- Otherwise, if the data type of one input is a fixed-point type, the data type of the output is the smallest fixed-point type capable of representing the result without any loss of precision.
- Otherwise, if the data type of both inputs is Boolean the output is Boolean as well.
- Finally, if one input is integral and the other is also integral or Boolean, the output is integral. If both inputs are unsigned the output is unsigned, otherwise it is signed. The bit width of the output is largest among the bit widths of the inputs.

Parameters

Saturate on integer overflow

This parameter specifies whether integer overflow is handled by wrapping (default) or by saturating. This parameter is relevant only if the output is integral (int8, int16, int32, uint8, uint16, uint32).

Settings for the **Saturate on integer overflow** parameter are:

Table 114: **Saturate On Integer Overflow Parameter**

Setting	Description
Not selected	Integer overflow is handled by wrapping.
Selected	Integer overflow is handled by saturation.

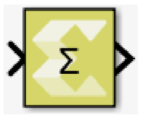
When overflow is detected, the Diagnostic Viewer displays messages that depend on the diagnostic action you specify in the Simulink Editor. To configure, select **Simulation → Model Configuration Parameters → Diagnostics → Data Validity** for your model in the Simulink Editor, then set the **Wrap on overflow** or **Saturate on overflow** parameter.

Sum of Elements

Perform element-wise addition on the input, column-wise, row-wise, or in all dimensions

Library

Math Functions / Math Operations



Description

This block performs element-wise addition on a vector or matrix type input. If input is a scalar then this block operates as a pass-through. The output is a scalar type if input is a vector or a matrix and the **Sum over** parameter is set as **All dimensions**.

Data Type Support

Data type support for the block is:

- The Sum of Elements block supports any floating-point, fixed-point, integer or Boolean data type.

- The output type is a scalar or a vector depending on the dimensions of the input and the selection of the **Sum over** parameter.
- The block can perform element-wise addition on real or complex number data.

Parameters

Sum over

The **Sum over** parameter value is used to decide whether elements will be added in all dimensions or in one of the dimensions.

Following are settings for the **Sum over** parameter.

Table 115: **Sum Over Parameter**

Setting	Description
All dimensions	Add all elements of the input signal (output is scalar)
Specified dimension	This option shows an edit box, **Dimension**, where the specific dimension value can be entered.

Dimension

The **Dimension** parameter is displayed only if the **Sum over** parameter value is set to **Specified dimension**.

Settings for the **Dimension** parameter are:

Setting	Description
1	Add input over row dimension. Output is a row matrix.
2	Add input over column dimension. Output is a column matrix.

Tangent

Perform an element-wise computation of the tangent function for the given argument

Library

Math Functions / Math Operations

Description

The block returns the output of the function tan(`x`) for each element in array `x`.

Data Type Support

Data types accepted at the input of the block are:

- Dimension : Input can be scalar, vector or matrix.
- Data Types: Input supports signals of integer type, floating point type (double, single and half) and fixed-point type.
- Complex Number Support: No

  Output has the same dimension and type as the input.

Parameters

The Tangent block has no parameters to set.

Transpose

Perform an element-wise transpose operation on the input signal

Library

Math Functions / Matrices and Linear Algebra

Description

The Transpose block performs a transpose operation on the input signal.

Data Type Support

This block supports all data types supported by Vitis Model Composer. The input signal can be real or a complex number of scalar, vector, or matrix type. The output type is always the same as that of the input.

Parameters

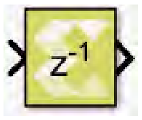
The Transpose block has no parameters to set.

Unit Delay

Provides a delay of one sample period

Library

Signal Operations



Description

The Unit Delay block provides a delay of one sample period. The output has the same sample time as the input. The output dimension is the same as the initial condition dimension if the input is scalar. Otherwise, the output dimension is the same as the input dimension and should match the dimension of the initial condition.

Data Type Support

Data type support is:

- All data types are supported.
- Input can be a vector or a matrix. If input is a vector or a matrix and the initial value is a scalar, the scalar value will apply to all the elements of the input during the first cycle.
- Output is complex if the input is complex.
- If the initial value is complex but the input signal is type real, the block gives an error indicating that the input type must be complex.

Parameters

Initial Condition

Specifies the initial value.

The **Initial Condition** can be scalar, vector, or matrix, of real or complex type.

Window Processing

Assemble an output matrix by applying the kernel subsystem to submatrices (windows) of the input matrix in row-major order

Library

Ports and Subsystems

Description

The Window Processing block is a masked subsystem for assembling an output matrix by applying a kernel subsystem to sub matrices of the input matrix in row-major order. You customize the Window Processing block by specifying its parameters and by adding blocks to the Kernel subsystem for computing a scalar element of the output in terms of the corresponding input window. The computation can be thought of to proceed in the following steps:

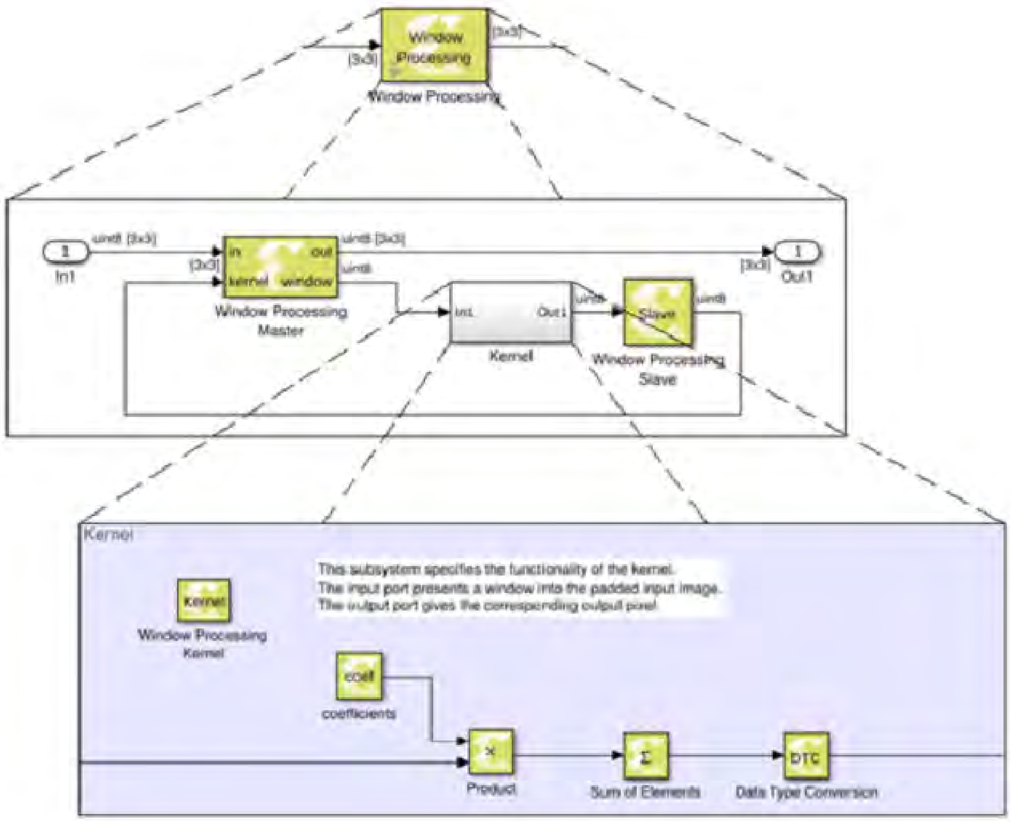
1. Compute the padded input matrix depending on the setting of the **Output size** parameter.

   Let the dimensions of the input be Min,Nin; let the dimensions of the window be Mwin, Nwin.

   - If the **Output size** is set to **Valid**, no padding is performed, and the dimensions of the output are Min-Mwin+1,Nin-Nwin+1.
   - If the **Output size** is set to **Same as input**, the input is padded with Mwin-1 rows and Nwin-1 columns of 0s. Half of rows added is put at the top and the remainder goes at the bottom. If Nwin is even then the bottom gets one more row of padding than the top. The same is done for adding the columns. The size of the output is the same as the size of the input.
   - If the **Output size** is set to **Full**, the input is padded with 2*(Mwin-1) rows and 2*(Nwin-1) columns of 0s. Half of the padding rows (columns) are added at the top (left) and the other half at the bottom (right).
2. For each element (i,j) of the output (iterate over output in row-major order):
   - Select the MxN sub matrix from the padded input matrix starting with element (i,j)
   - Apply the kernel subsystem to this sub matrix
   - Assign the scalar output of the kernel to element (i,j) of the output.

The hierarchy of the Window Processing block is shown below.

Figure 418: **Hierarchy of Window Processing Block**



Restrictions on the use of the Window Processing block are:

- The topology of the window processing subsystem shall not be modified. It must match the topology show in the figure above.
- The input signal must be a matrix.
- The kernel must have precisely one input port and one output port.
- The output of the kernel must be scalar.
- The input of the kernel must have the dimensions specified by the window size parameter.
- The dimensions of the input must be at least as large as the window size.
- The kernel must not contain any of the following blocks:
  - Blocks that have internal state, such as the Unit Delay block.
  - The following Digital Signal Processing blocks: FFT and IFFT.
  - Blocks created with the `xmcImportFunction` command.
- The kernel must not contain an if-action subsystem.

Data Type Support

There are no restrictions on the data types of the input or output signals.

Parameters

Window size

This parameter specifies the size of the window. Enter a 2-element vector of real positive integers, for example [5,3], for the **Window size**. Please note that **Window size** cannot specify more than a total of 255 elements.

Output size

This parameter specifies how the edges of the input image are treated.

Let Min, Nin be the dimensions of the input, and let Mwin, Nwin be the **Window size**. The dimensions of the output are as follows:

- Min+Mwin-1, Min+Nwin-1 if **Full** is selected for **Output size**.
- Min, Nin if **Same as Input** is selected for **Output size**.
- Min-Mwin+1, Min-Nwin+1 if **Valid** is selected for **Output size**.

Supported Simulink Blocks

You can mix blocks from the HLS block library and blocks from Simulink®, and other add-on toolboxes, during simulation in Simulink. However, you can only generate output from Model Composer from the top-level subsystem that uses only a limited set of Simulink blocks that are supported for code generation.

The following Simulink blocks are fully supported by Model Composer for code generation, and can be found in the Vitis Model Composer HLS block library as well.

Table 116: **Supported Simulink Blocks**

Simulink Block	Description
Action Port	Place this block in a subsystem to link to a signal from an `If` block or a `Switch-Case` block.
Bus Creator	This block creates a bus signal from multiple inputs.
Bus Selector	Pulls signals from an input bus to pass to output.
Display	Provides numeric display of input values.
DocBlock	Create and edit text associated with a model, and save that text with the model.
From	Receive signals from the Goto block with the specified tag.
Goto	Send signals to From blocks that have the specified tag.
If	Provides an `IF`/`ELSEIF`/`ELSE` condition for branching inputs to alternate outputs.

Table 116: **Supported Simulink Blocks** *(cont'd)*

Simulink Block	Description
Inport	Provide an input port for a subsystem or model.
Merge	Merge multiple input signals into a single output signal whose initial value is specified by the 'Initial output' parameter.
Outport	Provide an output port for a subsystem or model.
Scope	Displays time domain signals with respect to simulation time.
Stop Simulation	Stop simulation when input is non-zero.
Terminator	Used to "terminate" output signals to prevent warnings about unconnected output ports.
To File	Incrementally write data into a variable in the specified MAT-file.
To Workspace	Write input to specified timeseries, array, or structure in a workspace.

Refer to the Simulink documentation for a complete description of the block.

AI Engine Blockset

***Note*:** The AIE FIR Filter block which was part of AI Engine/DSP library will be deprecated in a future release. Please replace all instances of this block in the model with equivalent blocks from the Xilinx Toolbox/AI Engine/DSP library.

AIE Class Kernel

This block allows you to import class-based kernels.

Library

AI Engine/User-Defined Functions

Description

The AIE Class Kernel block is used to import class-based kernels. This block also supports importing class templates to define a family of kernels.

Parameters

Parameter Name	Parameter Type	Criticality	Description
Kernel header file	String	Mandatory	Name of the header file that contains the kernel class and registerKernelClass method declarations The string could be just the file name, a relative path to the file or an absolute path of the file. Use the browse button to choose the file.
Kernel class	String	Mandatory	Name of the kernel class which contains member variables and kernel member functions.
Kernel function	String	Mandatory	Name of the kernel member function for which the block is to be created. This function should be registered using the registerKernelClass method in the kernel header file.
Kernel source file	String	Mandatory	Name of the source file that contains where the kernel member function definition and non-default constructor parameter values are specified. The string could be the file name, a relative path to the file or an absolute path of the file.
Kernel search paths	Vector of strings	Optional	If the kernel header file or the kernel source file are not found using the value provided through the 'Kernel header file' or 'Kernel source file' fields respectively, then the paths provided through 'Kernel search paths' are used to find the files. This parameter allows use of environment variables while specifying paths for the kernel header file and the kernel source file. The environment variable can be used in either `${ENV}` or `$ENV` format.
Preprocessor options		Optional	Optional preprocessor arguments for downstream compilation with specific preprocessor options. The following two preprocessor option formats are accepted and multiple can be selected: `-Dname` and `-Dname=definition` separated by a comma. That is, the optional argument must begin with `-D` and if the option definition value is not provided, it is assumed to be `1`.

AIE Graph

This block allows you to import an AI Engine graph.

Library

AI Engine/User-Defined Functions

Description

The AIE Graph block allows you to import an AI Engine program that consists of a dataflow graph specification written in C++.

Parameters

- **Graph File:** Specifies whether the graph import should be done using the header file (`*.h`) or using source file (`*.cpp`). By default the header file is selected.
- **Graph Header file(*.h):** This is the mandatory string that specify the file (`.h`), where the application graph class is defined and the Adaptive Data Flow (ADF) header (`adf.h`), kernel function prototypes are included. This parameter is only visible when you choose the **header file** (`*.h`) option in the graph file.
- **Graph Source file (*.cpp):** This is the mandatory string that specify the file(.cpp), where the adf dataflow graph is instantiated. This file should contain the main() function, from where the dataflow graph initializes and runs. This parameter is only visible when you choose the 'source file (*.cpp) option in the Graph file. This option is available only when the Source file (*.cpp) is selected in Graph file.
- **Graph Class:** This is a mandatory string that specifies the name of the graph class. This parameter is only visible when you choose the **header file** (*.h) option in the graph file.
- **Graph Search paths:** This is a vector of strings that specifies the search paths where header files, kernels, and other include files can be found and included for simulation. The search path $XILINX_VITIS/adf/include (where adf.h is defined) is included by default and does not need to be specified.
- **Preprocessor options:** This is an optional parameter and should be specified with a preprocessor argument for downstream compilation with specific preprocessor options. The following preprocessor option formats are accepted and multiple can be selected: '`-Dname`' and '`-Dname=definition`'. That is, the optional argument must begin with the '`-D`' string and if the option definition value is not provided, it is assumed to be 1.

AIE Kernel

This block allows you to import an AI Engine kernel.

Library

AI Engine/User-Defined Functions

Description

The AIE Kernel block allows you to import an AI Engine kernel which is a C/C++ program. This block supports importing Window, Stream, Cascade, and Run time parameter as arguments to kernel function. This block also allows you to import a function template with typename template parameter T, and a non-type (integral) template parameter N.

Parameters

Parameter Name	Parameter Type	Criticality	Description
Kernel header file	String	Mandatory	Name of the header file that contains the kernel function declaration. The string could be just the file name, a relative path to the file or an absolute path of the file. Use the browse button to choose the file.
Kernel function	String	Mandatory	Name of the kernel function for which the block is to be created. This function should be declared in the kernel header file.
Kernel init function	String	Optional	Name of the initialization function used by the kernel function.
Kernel source file	String	Mandatory	Name of the source file that contains the kernel function definition. The string could be the file name, a relative path to the file or an absolute path of the file.
Kernel search paths	Vector of Strings	Optional	If the kernel header file or the kernel source file are not found using the value provided through the 'Kernel header file' or 'Kernel source file' fields respectively, then the paths provided through 'Kernel search paths' are used to find the files. This parameter allows use of environment variables while specifying paths for the kernel header file and the kernel source file. The environment variable can be used in either `${ENV}` or `$ENV` format.
Preprocessor options		Optional	Optional preprocessor arguments for downstream compilation with specific preprocessor options. The following two preprocessor option formats are accepted and multiple can be selected: `-Dname` and `-Dname=definition` separated by a comma. That is, the optional argument must begin with `-D` and if the option definition value is not provided, it is assumed to be `1`.

AIE Signal Spec

This block is used to specify various properties on signals within, as well as at the boundary of an AI Engine subsystem.

AIE Signal Spec

Library

AI Engine/Tools

Description

The AIE Signal Spec block allows you to specify the Platform IO (PLIO) width and FIFO depth value within the AI Engine subsystem.

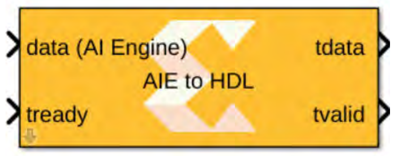
- Specifying the PLIO width at the boundary of the AI Engine subsystem will affect the throughput of data from the AI Engine domain to the programmable logic (PL) domain.
- Specifying the FIFO depth value can help avoid deadlock or stalling by creating more buffering in the paths.

Parameters

- **Connection Tab:**
  - **FIFO Depth (32-bit words):** Should be a positive integer value and the default value is '0'.
- **Platform I/O Tab:**
  - **PLIO Width:** Only auto, 32, 64, and 128 are possible values. 'auto' is the default value.
  - **Specify PLIO frequency:** When set to **ON**, you can specify the Programmable logic frequency in MHz. The default value is 250 MHz. In general, choose a reasonable target frequency depending on the complexity of the algorithm implemented.

AIE to HDL

This block is used to connect the input port of an HDL block with the output port of an AI Engine kernel or AI Engine graph block using an AXI4-Stream interface.



Library

AI Engine/Interfaces

Description

This block provides an interface between the AI Engine and HDL blocks.

- Input to the AIE to HDL block is a variable size signal (data) from AI Engine blocks along with the `tready` signal which indicates that the consumer can accept a transfer.
- Output from the AIE to HDL block is `tdata` and `tvalid` that indicates the producer has valid data available. A transfer takes place when both `tvalid` and `tready` are asserted.

Parameters

- **Output Data Type:** The following table shows different Output data types that are supported by AIE to HDL blocks and the corresponding input data type to the block.

Output Data Type	Input to AIE - HDL Block
int32	int32
uint32	int8, uint8, int16, uint16, uint32, float, cint16
sfix64	int64
ufix64	int8, uint8, int16, uint16, cint16, int32, uint32, cint32, uint64, float, float(c)
ufix128	int8, uint8, int16, uint16, cint16, int32, uint32, cint32, int64, uint64, float, float(c)

- **Output Sample Time:** Set the `Output Sample Time` to (Input Sample Period)/(Input Size)/ii.

***Note*:** For more information on setting this block and examples, refer to GitHub.

AIE to HLS

This block is used to connect an input port of an HLS kernel block to the output port of an AI Engine block in cases where the data type or complexity of the ports involved do not match.

AIE to HLS Kernel

Library

AI Engine/User Defined Functions

Description

The AIE to HLS kernel block reformats a signal driven by an AI Engine block so that the resulting signal matches the data type and complexity required by the input of an HLS Kernel block.

Parameters

- **HLS Kernel Input Type:** Possible values are: ap_axis<32>, ap_axis<64>, ap_axis<128>, ap_axiu<32>, ap_axiu<64>, ap_axiu<128>, ap_int<32>, ap_int<64>, ap_uint<32>, ap_uint<64>,int, cint16, cint32, float, cfloat, long long, unsigned, unsigned long long
- **Output Size:** The size of the output port. The output port is a variable sized signal whose maximum size is specified by the OutputSize parameter. Default Output Size is '1'.

***Note*:**

1. The input data type must be one of the following: int8, int16, int32, int64, x_sfix128, uint8, uint16, uint32, uint64, x_ufix128.
2. The input can be real or complex, but complex inputs are supported only for int16 and int32.

DDS

AIE DDS
out
DDS

Library

AI Engine/DSP

Description

This block implements the Direct Digital Synthesizer (DDS) targeted for AI Engines.

Parameters

- **Main:**
  - **Output data type:** Describes the type of individual data samples output of the DDS function. It should only be `cint16`.
  - **Output window size (Number of samples):** Specifies the number of samples in output window. The value must be in the range 8 to 1024 and the default value is `32`.

- **Phase increment :** Specifies the phase increment between samples. The value must be in the range `0` to `2^31` and the default value is `0`. Input value `2^31` corresponds to Pi (i.e., `180`).

  Phase increment is calculated using the formula (Fo*(2^N)) / Fs

  Where:

  - `Fo` = Output frequency
  - `N` = 32, which represents the accumulator width, and it is fixed
  - `Fs` = Sampling frequency
- **Sample time:** Specifies the sample time for the block output port. The default value is `-1`.

- **Advanced:**
  - **Target output throughput (MSPS):** Specifies the output sampling rate of the DDS function in Mega Samples per Second (MSPS). The value must be in the range `1` to `1000` and the default value is `200`.

FFT

Library

AI Engine/DSP

Description

This block implements the FFT targeted for AI Engines which use rounding method and saturates the output samples on overflow.

Parameters

- **Main:**
  - **Input/Output data type:** Describes the type of individual data samples input to and output from the filter function. Supported types are cint16, cint32, and cfloat.

- **FFT size:** This is an unsigned integer which describes the point size of the transformation. This must be 2^N, where N is in the range 4 to 11 inclusive. However, for `cint16` datatype, the FFT size can be 2^12, provided the FFT receives and outputs data to/from kernels on the same processor.

  Note: To understand more on achieving the 4096 point size FFT, refer to the AI Engine examples in GitHub.

- **Input Window Size(Number of Samples):** Describes the number of samples used as an input to the FFT.
- **Scale output down by 2^:** Describes the power of 2 to scale the result by prior to output.

- **Advanced:**
  - **Target input throughput:** Specifies the rate at which data samples should be processed. The default value is `200`.
  - **Specify the number of cascade stages:** When this option is not enabled, the tool will determine the filter configuration that best achieves the specified input sampling rate. When this option is enabled and the 'Number of cascade stages' is specified, the tool will guarantee the same. In such cases, however, the specified sample rate constraint may not be achieved.
  - **Number of cascade stages:** This determines the number of kernels the FFT will be divided over in series to improve throughput.

FIR Asymmetric Decimation

AIE FIR
Asymmetric Decimation

Library

AI Engine/DSP

Description

This block implements the FIR Asymmetric Decimation filter targeted for AI Engines.

Parameters

- **Main:**
  - **Input/Output data type:** Describes the type of individual data samples input to and output from the filter function. int16, cint16, int32, cint32, float, cfloat.

- **Filter coefficients data type:** Describes the type of individual coefficients of the filter taps. It should be one of int16, cint16, int32, cint32, float, cfloat and must also satisfy the following rules:
  - Complex types are only supported when the Input/Output data type is also complex.
  - 32-bit types are only supported when the Input/Output data type is also a 32-bit type.
  - Filter coefficients data type must be an integer type if the Input/Output data type is an integer type.
  - Filter coefficients data type must be a float type if the Input/Output data type is a float type.
- **Specify filter coefficients via input port:** When this option is enabled, the tool allows you to specify filter coefficients via the input port.
- **Filter coefficients:** Specifies the asymmetric filter coefficients. The filter length must be in the range 4 to 240 and must be an integer multiple of the decimation factor.
- **Decimation factor:** An unsigned integer which describes the decimation factor of the filter. It must be in the range 2 to 7.
- **Input window size (Number of samples):** Describes the number of samples used as an input to the filter function. The number of values in the output window will be (Input window size/decimation factor). The input window size must be an integer multiple of the decimation factor and the resulting output window size must be a multiple of 256 bits.
- **Scale output down by 2^:** Describes the power of 2 shift down applied to the accumulation of FIR terms before output. It must be in the range 0 to 61.
- **Rounding mode:** Describes the selection of rounding to be applied during the shift down stage of processing. The rounding options are as follows:
  1. Floor (truncate)
  2. Ceiling
  3. Round to positive infinity
  4. Round to negative Infinity
  5. Round symmetrical to Infinity
  6. Round symmetrical to zero
  7. Round convergent to even
  8. Round convergent to odd

  Modes 2 to 7 round to the nearest integer. They differ only in how they round for the value of 0.5.

- **Advanced:**

- **Target input throughput (MSPS):** Specifies the rate at which data samples should be processed. The default value is `200`.
- **Specify the number of cascade stages:** When this option is not enabled, the tool will determine the filter configuration that best achieves the specified input sampling rate. When the option is enabled, the 'Number of cascade stages' can be specified (which describes the number of AI Engine processors to split the operation over). However, this allows resources to be traded for higher performance and the specified input sampling rate constraint may not be achieved. The value must be in the range 1 to 9.

FIR Asymmetric Filter

Library

AI Engine/DSP

Description

This block implements the Single Rate Asymmetric FIR Filter targeted for AI Engines.

Parameters

- **Main:**
  - **Input/Output data type:** Describes the type of individual data samples input to and output from the filter function. int16, cint16, int32, cint32, float, cfloat.
  - **Filter coefficients data type:** Describes the type of individual coefficients of the filter taps. It should be one of int16, cint16, int32, cint32, float, or cfloat and must also satisfy the following rules:
    - Complex types are only supported when the Input/Output data type is also complex.
    - 32-bit types are only supported when the Input/Output data type is also a 32-bit type.
    - Filter coefficients data type must be an integer type if the Input/Output data type is an integer type.
    - Filter coefficients data type must be a float type if the Input/Output data type is a float type.
  - **Specify filter coefficients via input port:** When this option is enabled, the tool allows you to specify reloadable filter coefficients via the input port.

- **Filter coefficients :** This field should be specified with the asymmetric filter coefficients and must be in the range 4 to 240 inclusive.
- **Input window size (Number of samples):** Describes the number of samples used as an input to the filter function. The number of values in the output window will be equal to input window size also by virtue of the single rate nature of this function.
- **Scale output down by 2^:** Describes the power of 2 shift down applied to the accumulation of FIR terms before output. It must be in the range 0 to 61.
- **Rounding mode:** Describes the selection of rounding to be applied during the shift down stage of processing. The rounding options are as follows:

  1. Floor (truncate)
  2. Ceiling
  3. Round to positive infinity
  4. Round to negative Infinity
  5. Round symmetrical to Infinity
  6. Round symmetrical to zero
  7. Round convergent to even
  8. Round convergent to odd

  Modes 2 to 7 round to the nearest integer. They differ only in how they round for the value of 0.5.

- **Advanced:**
  - **Target input throughput (MSPS):** Specifies the rate at which data samples should be processed. The default value is `200`.
  - **Specify the number of cascade stages:** When this option is not enabled, the tool will determine the filter configuration that best achieves the specified input sampling rate. When the option is enabled, the 'Number of cascade stages' can be specified (which describes the number of AI Engine processors to split the operation over). However, this allows resource to be traded for higher performance and the specified input sampling rate constraint may not be achieved. The value must be in the range 1 to 9.

FIR Fractional Interpolation

Library

AI Engine/DSP

Description

This block implements the FIR Fractional Asymmetric Interpolation filter targeted for AI Engines.

Parameters

- **Main:**
  - **Input/Output data type:** Describes the type of individual data samples input to and output from the filter function. int16, cint16, int32, cint32, float, cfloat.
  - **Filter coefficients data type:** Describes the type of individual coefficients of the filter taps. It should be one of int16, cint16, int32, cint32, float, cfloat and must also satisfy the following rules:
    - Complex types are only supported when the Input/Output data type is also complex.
    - 32-bit types are only supported when the Input/Output data type is also a 32-bit type.
    - Filter coefficients data type must be an integer type if the Input/Output data type is an integer type.
    - Filter coefficients data type must be a float type if the Input/Output data type is a float type.
  - **Specify filter coefficients via input port :** When this option is enabled, the tool allows you to specify reloadable filter coefficients via input port.
  - **Filter coefficients:** This field should be specified with the asymmetric filter coefficients and must be in the range 4 to 240 inclusive.
  - **Interpolation factor:** An unsigned integer which describes the Interpolation factor of the filter. It must be in the range 3 to 16.
  - **Decimation factor:** An unsigned integer which describes the decimation factor of the filter. It must be in the range 2 to 16. The decimation factor should be less that the interpolation factor and should not be divisible by the interpolation factor.
  - **Input window size (Number of samples):** Describes the number of samples used as an input to the filter function. The number of values in the output window will be the input window size multipled by the Interpolation factor and divided by the decimation factor. In this instance it wouldresult in a fraction number of output samples which would be rounded down.
  - **Scale output down by 2^ :** Describes the power of 2 shift down applied to the accumulation of FIR terms before output. It must be in the range 0 to 61.

- **Rounding mode:** Describes the selection of rounding to be applied during the shift down stage of processing. The rounding options are as follows:

    1. Floor (truncate)
    2. Ceiling
    3. Round to positive infinity
    4. Round to negative infinity
    5. Round symmetrical to infinity
    6. Round symmetrical to zero
    7. Round convergent to even
    8. Round convergent to odd

    Modes 2 to 7 round to the nearest integer. They differ only in how they round for the value of 0.5.

- **Advanced:**
    - **Number of cascade stages:** This determines the number of kernels over which the function will be split. A higher number of cascade stages will result in higher throughput at the expense of resources. The value must be in the range 1 to 9.

FIR Halfband Decimator

Library

AI Engine/DSP

Description

This block implements the FIR Halfband Decimator targeted for AI Engines.

Parameters

- **Main:**
    - **Input/Output data type :** Describes the type of individual data samples input to and output from the filter function. int16, cint16, int32, cint32, float, cfloat.

- **Filter coefficients data type:** Describes the type of individual coefficients of the filter taps. It should be one of int16, cint16, int32, cint32, float, cfloat and must also satisfy the following rules:
  - Complex types are only supported when the Input/Output data type is also complex.
  - 32-bit types are only supported when the Input/Output data type is also a 32-bit type.
  - Filter coefficients data type must be an integer type if the Input/Output data type is an integer type.
  - Filter coefficients data type must be a float type if the Input/Output data type is a float type.
- **Specify filter coefficients via input port:** When this option is enabled, the tool allows you to specify reloadable filter coefficients via input port.
- **Filter coefficients:** Specifies the filter coefficients as a vector of $(N+1)/4+1$ elements, where 'N' is a positive integer that represents the filter length and must be in the range 4 to 240 inclusive.
- **Input window size (Number of samples):** Describes the number of samples used as an input to the filter function. The number of values in the output window will be the Input window size divided by two by virtue of the halfband decimation factor.
- **Scale output down by 2^:** Describes the power of 2 shift down applied to the accumulation of FIR terms before output. It must be in the range 0 to 61.
- **Rounding mode:** Describes the selection of rounding to be applied during the shift down stage of processing. The rounding options are as follows:
  1. Floor (truncate)
  2. Ceiling
  3. Round to positive infinity
  4. Round to negative infinity
  5. Round symmetrical to infinity
  6. Round symmetrical to zero
  7. Round convergent to even
  8. Round convergent to odd

  Modes 2 to 7 round to the nearest integer. They differ only in how they round for the value of 0.5.

- **Advanced:**
  - **Target input throughput (MSPS):** Specifies the rate at which data samples should be processed. The default value is `200`.

- **Specify the number of cascade stages :** When this option is not enabled, the tool will determine the filter configuration that best achieves the specified input sampling rate. When the option is enabled, the 'Number of cascade stages' can be specified (which describes the number of AI Engine processors to split the operation over). However, this allows resource to be traded for higher performance and the specified input sampling rate constraint may not be achieved. The value must be in the range 1 to 9.

FIR Halfband Interpolator

Library

AI Engine/DSP

Description

This block implements the FIR Halfband Interpolator targeted for AI Engines.

Parameters

- **Main:**
  - **Input/Output data type:** Describes the type of individual data samples input to and output from the filter function. int16, cint16, int32, cint32, float, cfloat.
  - **Filter coefficients data type:** Describes the type of individual coefficients of the filter taps. It should be one of int16, cint16, int32, cint32, float, cfloat and must also satisfy the following rules:
    - Complex types are only supported when the Input/Output data type is also complex.
    - 32-bit types are only supported when the Input/Output data type is also a 32-bit type.
    - Filter coefficients data type must be an integer type if the Input/Output data type is an integer type.
    - Filter coefficients data type must be a float type if the Input/Output data type is a float type.
  - **Specify filter coefficients via input port:** When this option is enabled, the tool allows you to specify reloadable filter coefficients via the input port.
  - **Filter coefficients:** Specifies the filter coefficients as a vector of $(N+1)/4+1$ elements, where 'N' is a positive integer that represents the filter length and must be in the range 4 to 240 inclusive.

- **Input window size (Number of samples)::** Describes the number of samples used as an input to the filter function. The number of values in the output window will be the Input window size multiplied by two by virtue of the halfband interpolation factor.
- **Scale output down by 2^:** Describes the power of 2 shift down applied to the accumulation of FIR terms before output. It must be in the range 0 to 61.
- **Rounding mode:** Describes the selection of rounding to be applied during the shift down stage of processing. The rounding options are as follows:

  1. Floor (truncate)
  2. Ceiling
  3. Round to positive infinity
  4. Round to negative infinity
  5. Round symmetrical to infinity
  6. Round symmetrical to zero
  7. Round convergent to even
  8. Round convergent to odd

  Modes 2 to 7 round to the nearest integer. They differ only in how they round for the value of 0.5.

- **Advanced:**
  - **Target input throughput (MSPS):** Specifies the rate at which data samples should be processed. The default value is `200`.
  - **Specify the number of cascade stages:** When this option is not enabled, the tool will determine the filter configuration that best achieves the specified input sampling rate. When the option is enabled, the 'Number of cascade stages' can be specified (which describes the number of AI Engine processors to split the operation over). However, this allows resource to be traded for higher performance and the specified input sampling rate constraint may not be achieved. The value must be in the range 1 to 9.

FIR Interpolation

Library

AI Engine/DSP

Description

This block implements the FIR Asymmetric Interpolation filter targeted for AI Engines.

Parameters

- **Main:**
  - **Input/Output data type:** Describes the type of individual data samples input to and output from the filter function. int16, cint16, int32, cint32, float, cfloat.
  - **Filter coefficients data type:** Describes the type of individual coefficients of the filter taps. It should be one of int16, cint16, int32, cint32, float, cfloat and must also satisfy the following rules:
    - Complex types are only supported when the Input/Output data type is also complex.
    - 32-bit types are only supported when the Input/Output data type is also a 32-bit type.
    - Filter coefficients data type must be an integer type if the Input/Output data type is an integer type.
    - Filter coefficients data type must be a float type if the Input/Output data type is a float type.
  - **Specify filter coefficients via input port:** When this option is enabled, the tool allows you to specify reloadable filter coefficients via the input port.
  - **Filter coefficients:** Specifies the filter coefficients as a vector of $(N+1)/4+1$ elements, where 'N' is a positive integer that represents the filter length and must be in the range 4 to 240 inclusive.
  - **Interpolation factor:** An unsigned integer which describes the interpolation factor of the filter. It must be in the range 1 to 16.
  - **Input window size (Number of samples):** Describes the number of samples used as an input to the filter function. The number of values in the output window will be Input window size multiplied by interpolation factor.
  - **Scale output down by 2^:** Describes power of 2 shift down applied to the accumulation of FIR terms before output. It must be in range 0 to 61.
  - **Rounding mode:** Describes the selection of rounding to be applied during the shift down stage of processing. The rounding options are as follows:
    1. Floor (truncate)
    2. Ceiling
    3. Round to positive infinity
    4. Round to negative infinity
    5. Round symmetrical to infinity

6. Round symmetrical to zero
7. Round convergent to even
8. Round convergent to odd

Modes 2 to 7 round to the nearest integer. They differ only in how they round for the value of 0.5.

- **Advanced:**
  - **Target input throughput (MSPS) :** Specifies the rate at which data samples should be processed. The default value is `200`.
  - **Specify the number of cascade stages:** When this option is not enabled, tool will determine the filter configuration that best achieves the specified input sampling rate. When the option is enabled and the 'Number of cascade stages' is specified, the tool will guarantee the same. In such cases, however, the specified input sampling rate constraint may not be achieved.

FIR Symmetric Decimation

Library

AI Engine/DSP

Description

This block implements the FIR Symmetric Decimation Filter targeted for AI Engines.

Parameters

- **Main:**
  - **Input/Output data type:** Describes the type of individual data samples input to and output from the filter function. int16, cint16, int32, cint32, float, cfloat.
  - **Filter coefficients data type:** Describes the type of individual coefficients of the filter taps. It should be one of int16, cint16, int32, cint32, float, cfloat and must also satisfy the following rules:
    - Complex types are only supported when the Input/Output data type is also complex.
    - 32-bit types are only supported when the Input/Output data type is also a 32-bit type.

    - Filter coefficients data type must be an integer type if the Input/Output data type is an integer type.
    - Filter coefficients data type must be a float type if the Input/Output data type is a float type.

- **Specify filter coefficients via input port:** When this option is enabled, the tool allows you to specify reloadable filter coefficients via the input port.
- **Filter coefficients:** This field should only be supplied for the first half of the filter length plus the center tap for odd lengths i.e., taps[] = {c0, c1, c2, ..., cN [, cCT]} where N = (FILTER_LENGTH)/2 and cCT is the center tap where FILTER_LENGTH is odd. For example, a 7-tap filter might use coeffs (1, 3, 2, 5, 2, 3, 1). This could be input as taps[]= {1,3,2,5} because the context of symmetry allows the remaining coefficients to be inferred.
- **Filter length:** This is an unsigned integer which describes the number of taps in the filter. The filter length must be in the range 4 to 240 and must be an integer multiple of the decimation factor.
- **Decimation factor:** An unsigned integer which describes the decimation factor of the filter. It must be in the range 2 to 3. For larger factors, use the FIR Asymmetric decimation filter.
- **Input window size (Number of samples):** Describes the number of samples used as an input to the filter function. The number of values in the output window will be the input window size divided by decimation factor by virtue of the decimation factor.
- **Scale output down by 2^:** Describes power of 2 shift down applied to the accumulation of FIR terms before output. It must be in range 0 to 61.
- **Rounding mode:** Describes the selection of rounding to be applied during the shift down stage of processing. The rounding options are as follows:

    1. Floor (truncate)
    2. Ceiling
    3. Round to positive infinity
    4. Round to negative infinity
    5. Round symmetrical to infinity
    6. Round symmetrical to zero
    7. Round convergent to even
    8. Round convergent to odd

    Modes 2 to 7 round to the nearest integer. They differ only in how they round for the value of 0.5.

- **Advanced:**

- **Target input throughput (MSPS):** Specifies the rate at which data samples should be processed. The default value is `200`.
- **Specify the number of cascade stages:** When this option is not enabled, the tool will determine the filter configuration that best achieves the specified input sampling rate. When the option is enabled and the 'Number of cascade stages' is specified, the tool will guarantee the same. In such cases, however, the specified input sampling rate constraint may not be achieved.

FIR Symmetric Filter

Library

AI Engine/DSP

Description

This block implements the Single Rate Symmetric FIR Filter targeted for AI Engines.

Parameters

- **Main:**
  - **Input/Output data type:** Describes the type of individual data samples input to and output from the filter function. int16, cint16, int32, cint32, float, cfloat.
  - **Filter coefficients data type:** Describes the type of individual coefficients of the filter taps. It should be one of int16, cint16, int32, cint32, float, cfloat and must also satisfy the following rules:
    - Complex types are only supported when the Input/Output data type is also complex.
    - 32-bit types are only supported when the Input/Output data type is also a 32-bit type.
    - Filter coefficients data type must be an integer type if the Input/Output data type is an integer type.
    - Filter coefficients data type must be a float type if the Input/Output data type is a float type.
  - **Specify filter coefficients via input port:** When this option is enabled, the tool allows you to specify reloadable filter coefficients via the input port.

- **Filter coefficients:** This field should only be supplied for the first half of the filter length plus the center tap for odd lengths i.e., taps[] = {c0, c1, c2, ..., cN [, cCT]} where N = (FILTER_LENGTH)/2 and cCT is the center tap where FILTER_LENGTH is odd. For example, a 7-tap filter might use coeffs (1, 3, 2, 5, 2, 3, 1). This could be input as taps[]= {1,3,2,5} because the context of symmetry allows the remaining coefficients to be inferred.
- **Filter length:** This is an unsigned integer which describes the number of taps in the filter.
- **Input window size (Number of samples):** Describes the number of samples used as an input to the filter function. The number of values in the output window will be the input window size by of virtue the single rate nature of this filter.
- **Scale output down by 2^:** Describes power of 2 shift down applied to the accumulation of FIR terms before output. It must be in range 0 to 61.
- **Rounding mode:** Describes the selection of rounding to be applied during the shift down stage of processing. The rounding options are as follows:

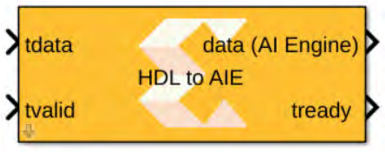
  1. Floor (truncate)
  2. Ceiling
  3. Round to positive infinity
  4. Round to negative infinity
  5. Round symmetrical to infinity
  6. Round symmetrical to zero
  7. Round convergent to even
  8. Round convergent to odd

  Modes 2 to 7 round to the nearest integer. They differ only in how they round for the value of 0.5.

- **Advanced:**
  - **Target input throughput (MSPS):** Specifies the rate at which data samples should be processed. The default value is `200`.
  - **Specify the number of cascade stages:** When this option is not enabled, the tool will determine the filter configuration that best achieves the specified input sampling rate. When the option is enabled and the 'Number of cascade stages' is specified, the tool will guarantee the same. In such cases, however, the specified input sampling rate constraint may not be achieved.

HDL to AIE

This block is used to connect the output ports of HDL blocks to the input ports of AI Engine blocks using the AXI4-Stream protocol.



Library

AI Engine/Interfaces

Description

This block provides an interface between the HDL and AI Engine blocks.

- Input to the HDL to AIE block is `tdata` which is the primary input for the data. The `tvalid` signal indicates that the producer has valid data.
- Output from the HDL to AIE block is a variable size signal (data) to AI Engine blocks along with the `tready` signal which indicates that the consumer can accept a transfer. A transfer takes place when both `tvalid` and `tready` are asserted.

Parameters

- **Output Data Type:** The following table shows the Output data types that are supported by the HDL to AIE blocks and the corresponding input data type to the block.

Output Data Type	Input to HDL - AIE Block
int8	uint32, ufix64, ufix128
uint8	uint32, ufix64, ufix128
int16	uint32, ufix64, ufix128
uint16	uint32, ufix64, ufix128
cint16	uint32, ufix64, ufix128
int32	int32, ufix64, ufix128
uint32	uint32, ufix64, ufix128
cint32	ufix64, ufix128
int64	sfix64
uint64	ufix64, ufix128
float	uint32, ufix64, ufix128
float(c)	ufix64, ufix128

- **Output Sample Time:** Set the `Output Sample Time` to:

$$output\ sample\ time = input\ sample\ time * \frac{output\ bit\ width}{input\ bit\ width}$$

***Note*:** For more information on setting this block and examples, refer to GitHub.

- **Samples per output frame:** This determines the number of samples to be queued in the buffer before the block updates the frame.
- **Tready Sample time:** This should be the same as the HDL design sample time.

HLS to AIE

This block is used to connect an input port of an AIE Kernel or AIE Graph block to an output port of an HLS Kernel block in cases where the datatype or complexities of the ports involved does not match.

HLS Kernel to AIE

Library

AI Engine/User Defined Functions

Description

The HLS Kernel to AIE block reformats a signal driven by a port of an HLS Kernel block so that the resulting signal matches the data type and complexity required by an AI Engine block.

Parameters

- **AIE Input Type:** Possible values are: int8, int16, int32, int64, uint8, uint16, uint32, uint64, cint16, cint32,float, cfloat

  ***Note*:** The inputs must be real and the supported input data types are: int32, int64, x_sfix128, uint32, uint64, x_ufix128.

- **Output Size:** The size of the output port. The output port is a variable sized signal whose maximum size is specified by the Output Size parameter. Default size is '1'.

HLS Kernel

This block lets you import an HLS kernel code with a streaming interface.

Library

AI Engine/User-Defined Functions

Description

The HLS Kernel block allows you to import an HLS kernel, which is a proper HLS IP that can be used in Vitis™ HLS and synthesized directly.

Parameters

Parameter Name	Parameter Type	Criticality	Description
Kernel header file	String	Mandatory	The name of the HLS kernel header file that contains the function declaration. The string could be just the file name, a relative path to the file or an absolute path of the file. Use the browse button to select the file.
Kernel function	String	Mandatory	The name of the kernel function in C/C++ for which HLS Kernel block is to be created.
Kernel source file	String	Mandatory	The name of the source file that contains the kernel function implementation (definition). The string could be just the file name, a relative path to the file or an absolute path of the file. Specifies the search path for source files (`.cc`, `.hpp`) from the MATLAB current folder.
Kernel search paths	Vector of Strings	Optional	If the kernel header file or the kernel source file is not found using the value provided through the 'Kernel header file' or 'Kernel source file' fields respectively, then the paths provided through 'Kernel search paths' are used to locate the files. This parameter allows use of environment variables while specifying paths for the kernel header file and the kernel source file. The environment variable can be used in either `${ENV}` or `$ENV` format.
Preprocessor Options		Optional	Optional preprocessor arguments for downstream compilation with specific preprocessor options. The following two preprocessor option formats will be accepted and multiple can be selected. `-Dname` and `-Dname=definition`. That is, the optional argument must begin with the `-D` string and if the option definition value is not provided, it is assumed to be `1`.

IFFT

Library

AI Engine/DSP

Description

This block implements the Inverse FFT targeted for AI Engines which use the rounding method and saturates the output samples on overflow.

Parameters

- **Main:**
    - **Input/Output data type:** Describes the type of individual data samples input to and output from the filter function. Supported types are cint16, cint32 and cfloat.
    - **IFFT size:** This is an unsigned integer which describes the point size of the transformation. This must be 2^N, where N is in the range 4 to 11 inclusive. However, for `cint16` datatype, the IFFT size can be 2^12, provided the IFFT receives and outputs data to/from kernels on the same processor.

      Note: To understand more on achieving the 4096 point size FFT, refer to the AI Engine examples in GitHub.
    - **Input Window Size (Number of Samples):** Describes the number of samples used as an input to the IFFT.
    - **Scale output down by 2^:** Describes the power of 2 to scale the result by prior to output.
- **Advanced:**
    - **Target input throughput:** Specifies the rate at which data samples should be processed. The default value is `200`.
    - **Specify the number of cascade stages:** When this option is not enabled, the tool will determine the filter configuration that best achieves the specified input sampling rate. When this option is enabled and the 'Number of cascade stages' is specified, the tool will guarantee the same. In such cases, however, the specified sample rate constraint may not be achieved.

- **Number of cascade stages:** This determines the number of kernels the FFT will be divided over in series to improve throughput.

Mixer

in1
AIE Mixer
out
Mixer

Library

AI Engine/DSP

Description

This block implements the Mixer targeted for AI Engines.

Parameters

- **Input/Output data type:** Describes the type of individual data samples input to and output from the Mixer function. It should only be `cint16`.
- **Input window size (Number of samples) :** Specifies the number of samples in the input window. The value must be in the range `16` to `4096`. Default value is `32`.
- **Mixer mode:** This specifies the mixer operation modes. Two modes are supported by Mixer function:
  - **Single Input Mode:** This is a DDS plus Mixer for a single data input port. Each data input sample is complex multiplied with the corresponding DDS sample to create a modulated signal that is written to the output window. This is the default Mixer mode.
  - **Dual Input Mode:** This is a special configuration for symmetrical carriers and two data input ports. Each data sample of the first input is complex multiplied with the corresponding DDS sample to create a modulated signal. Each data sample of the second data input is complex multiplied with the conjugate (which is equivalent to a signal rotating in the opposite direction) of the DDS sample to create a second modulated signal. These two modulated signals are added together and written to the output window.
- **Phase increment :** This specifies the phase increment between the samples. The value should be in the range `0` to `2^31`.

  Phase increment is calculated using the formula (Fo*(2^N))/ Fs)

Where:

- `Fo` = Output frequency
- `N` = 32, which represents the accumulator width, and it is fixed
- `Fs` = Sampling frequency

- **Target input throughput (MSPS):** Specifies the input sampling rate of the function in Mega Samples per Second (MSPS). The value must be in the range `1` to `1000` and the default value is `200`.

RTP Source

RTP Source
out

Library

AI Engine/Tools

Description

This block can be used as a source for the RTP input of an AI Engine block. When the RTP input is a scalar, the 'RTP Value' parameter should be a row vector. At each time step, the output is set to one of the elements of the vector starting with the first element. If an element of the vector is NaN, at the corresponding sampling time, the output will be an empty variable size signal.

If the RTP input is a vector, the 'RTP value' parameter should be a matrix. Each column represents an RTP input vector. A NaN column will produce an empty variable size signal output.

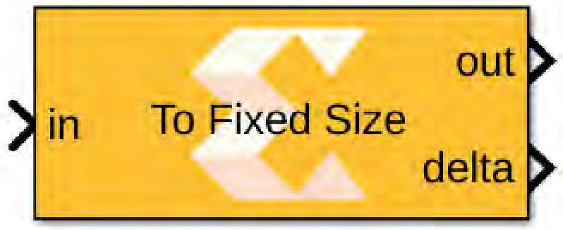
Parameters

- **RTP Value:** This represents the value which can be given as an input to an AI Engine block. This can be a scalar, vector, or a matrix and it accepts real or complex data.
- **Sample Time:** Specifies the interval between the times that the RTP source block output can change during simulation.
- **Form output after final data:** Represents a method to determine block output after the final data point.
  - **Empty:** This option sets the RTP block output to empty after final data.
  - **Holding Final value:** When this option is selected, block holds the final value.

- **Cyclic repetition:** This option repeats the RTP block data from first value.

To Fixed Size

This block takes a variable size vector as an input and produces a fixed size vector as output. The block copies samples from the input to the output.



Library

AI Engine/Tools

Description

The To Fixed Size block takes a variable size vector as an input and produces a fixed size vector as output. The block copies samples from the input to the output.

Parameters

- **Copy Method:** Specifies the method to copy the samples from input to the output:
  - **Pad with zeros or discard excess samples:** When Copy Method is set to this option, zeros will be padded in case of insufficient samples and excess samples will be discarded.
  - **Buffer samples to form output:** When Copy Method is set this option, the input will be buffered until the number of samples reaches the Output Size. The buffered samples will then be transferred to the out port, and the valid signal is set to true. When there are not enough samples buffered, the output port will be a vector of zeros, and the valid port is set to false.
- **Show delta between input and output sizes:** This is an optional port, available only when the "Pad with zeros or discard excess samples" Copy Method is selected. It shows the difference between the number of samples in the input variable size signal and the output size.
- **Output Size:** This specifies the size of the output port.
  - **Inherit : Same as Input:** If this option is enabled, the block will only accept the input data when the valid port is true.
  - **Specify Output Size:** When this option is selected, you can specify the value of the required output size.

To Variable Size

in
To Variable Size
out

Library

AI Engine/Tools

Description

This block takes a fixed sized vector input and produces a variable sized vector output. The maximum size of the output vector is specified by the Output Size parameter. If there is not enough samples to pack the output, the output will be an empty variable size signal.

Parameters

- **Show Valid Input:** If this option is enabled, the block will only accept the input data when the valid port is true.
- **Output Size:** This specifies the size of the output port.
  - **Inherit : Same as Input:** If this option is enabled, the block will only accept the input data when the valid port is true.
  - **Specify Output Size:** When this option is selected, you can specify the value of the required output size.

Variable Size Signal to Workspace

This block is used to save variable size signal data from your Simulink® simulation to the MATLAB® workspace.

Library

AI Engine/Tools

Description

AI Engine blocks produce variable signal outputs. The Variable Size Signal to Workspace block allows you to write data into theMATLAB workspace in a structured format.

***Note*:** This block behaves similarly to theSimulink To Workspace block but can only be connected to a variable size signal. The settings of the block can be accessed from Model settings (**Ctrl+E**).

Parameters

- **Variable Name:** Using this parameter, you can specify the name for workspace variable.

Appendix A

Model Composer Utilities and Programmatic Access

AI Engine Utilities

xmcLibraryPath	To set the root of the DSPLib from a Matlab command window to a custom location in your local directory.
xmcVitisRead	Reads Vitis data files and outputs them into MATLAB.
xmcVitisWrite	Takes an existing array and creates a file that can be read by AIEsimulator and x86simulator.

xmcLibraryPath

xmcLibraryPath is a MATLAB utility that sets the root of the DSPLib from a MATLAB command window to a custom location in your local directory, using the MATLAB utility `xmcLibraryPath`.

Syntax

```
xmcLibraryPath(‘<Arg1>’, ‘dsplib’, <Arg2>);
```

'Arg1' should be one of the options specified in the following list.

Arg1	Description
'set'	Set the DSPLib root to the custom location.
'reset'	Reset the DSPLib root to the default location.
'get'	To get the current DSPLib root location.
'getDefault	To get the default DSPLib root location.

Example 1

```
xmcLibraryPath(‘set’, ‘dsplib’, '/workspace/Github_Repo/Vitis_Libraries/
dsp');
```

This option sets the DSPlib to point to the code from GitHub repository.

Example 2

```
xmcLibraryPath('reset', 'dsplib');
```

This option sets the DSPLib to the Model Composer installed directory, which is the default location.

xmcVitisRead

xmcVitisRead is a MATLAB utility that reads data files and outputs them into MATLAB. The function supports real and complex, signed and unsigned numbers. If the file contains timestamps, they will be added to a separate, parallel array.

Syntax

```
A = xmcVitisRead(fileIn,dataType)
[A, TS] = xmcVitisRead(fileIn,dataType)
```

Where, the option `fileIn` is the file path and the `dataType` represents the datatype of the imported data.

The following table shows the list of datatypes and maximum possible columns allowed in the input file.

Table 117: **Data Types**

Data Types	Data Size (bits)	Maximum Possible Columns
int8	8	16
int16	16	8
int32	32	4
int64	64	2
uint8	8	16
uint16	16	8
uint32	32	4
uint64	64	2
cint16	32	4
cint32	64	2
float	32	4
cfloat	64	2

Example

Let `Input.txt` be a file of the following format containing real numbers:

```
"T 470 ns
             1651 -17 6646 -5720
             T 472 ns
             8850 -2469 2711 7752
             T 474 ns
             -4938 -6103 -4659 -2352
             T 476 ns
             -2144 -6453 1410 5685
             T 478 ns
             -1591 1962 1190 8775"
```

The following function call imports the array as MATLAB `int16` values.

```
→   A = xmcVitisRead("Input.txt",'int16')
```

This is the resulting column vector:

```
 A = [
                    1651,
                   -17,
                    6646,
                   -5720,
                    8850,
                   -2469,
                    2711,
                    7752,
                   -4938,
                   -6103,
                   -4659,
                   -2352,
                   -2144,
                   -6453,
                    1410,
                    5685,
                   -1591,
                    1962,
                    1190,
                    8775]
```

The following function call produces two arrays, signal and timestamp. The result is as follows.

```
→   [A, timestamp] = xmcVitisRead("Input.txt",'int16')

                  TS = [        A = [
                       470,          1651,
                       470,         -17,
                       470,          6646,
                       470,         -5270,
                       472,          8850,
                       472,         -2469,
                       472,          2711,
                       472,          7752,
                       474,         -4938,
                       474,         -6103,
                       474,         -4659,
                       474,          -2352,
                       476,          -2144,
                       476,          -6453,
                       476,           1410,
                       476,           5685,
                       478,          -1591,
                       478,           1963,
                       478,           1190,
                       478]           8775]
```

IMPORTANT! *The function checks for the number of columns and returns an error if the number of columns exceeds the theoretical maximum for that data type.*

IMPORTANT! *If the file contains timestamps but they are not required, the `timestamp` output can be left blank. Conversely, if the file does not have timestamps but two outputs are specified, the timestamp output will be an empty numeric array.*

xmcVitisWrite

This function takes an existing array and creates a file that can be read by AIEsimulator and x86simulator. The input array can be of any real or complex data type.

- Complex data types are automatically detected, therefore only the data type (for example,.`int16`) needs to be specified.
- The function also accepts the PLIO width and uses the data type to calculate the number of columns in the file. If the PLIO width is less than the size of the data type, the function will return an error.

Syntax

```
xmcVitisWrite(fileName,arrayIn,dataType,widthPLIO)
```

Where:

- `fileName` is the output file name.
- `arrayIn` is the Input array.
- `dataType` is the datatype of the data.
- `widthPLIO` is the PLIO width (must be 32, 64, or 128)

The following table shows the list of datatypes and maximum possible columns in the output file, based on the PLIO width.

Table 118: **Data Types**

Data Types	Data Size (bits)	Maximum Possible Columns
int8	8	16
int16	16	8

Table 118: **Data Types** *(cont'd)*

Data Types	Data Size (bits)	Maximum Possible Columns
int32	32	4
int64	64	2
uint8	8	16
uint16	16	8
uint32	32	4
uint64	64	2
cint16	32	4
cint32	64	2
float	32	4
cfloat	64	2

Example

Let A be an array of complex numbers of type `cfloat` and the PLIO width is set to 128.

```
A = [
                   3.142 + 1.463i,
                   6.288 + 3.079i,
                   3.333 + 1.493i,
                   3.781 + 8.781i,
                   3.142 + 1.463i]
```

Note: The data can be either a row or column vector.

The function call is as follows.

```
→ xmcVitisWrite("Output.txt",A,'cfloat',128);
```

The file will be saved as `Output.txt` in the working directory, as specified in the function call. The file content is as follows.

```
"3.142 1.463
              6.288 3.079
              3.333 1.493
              3.781 8.781
              3.142 1.463"
```

The columns are calculated automatically from the size of the data type and the PLIO width. A complex number is divided into two columns in the file, corresponding to real and imaginary parts in that order. A file with complex values will have at least two columns, while one with real values will have at least one.

HDL Utilities

xilinx.analyzer	Provides the interface between the Model Composer model and Vivado® timing paths.
xilinx.environment.getcachepath and xilinx.environment.setcachepath	Used to get and set the path Model Composer uses to store the simulation cache.
xilinx.resource_analyzer	Enables cross-probing between the Model Composer model and Vivado resource utilization data.
xilinx.utilities.importBD	Imports a BD file created in the Vivado IP integrator and creates a stub for the Model Composer model that is part of the design.
xlAddTerms	Automatically adds sinks and sources to Model Composer models.
xlConfigureSolver	Configures the Simulink® solver settings of a model to provide optimal performance during Model Composer simulation.
xlfda_denominator	Returns the denominator of the filter object in an FDATool block.
xlfda_numerator	Returns the numerator of the filter object in an FDATool block.
xlGenerateButton	Provides a programmatic way to invoke the Model Composer code generator.
xlgetparam and xlsetparam	Used to get and set parameter values in a HDL block.
xlgetparams	Used to get all parameter values in a HDL block.
xlGetReOrderedCoeff	The xlGetReOrderedCoeff function provides the re-ordered coefficient set of a FIR Compiler block.
xlOpenWaveFormData	Allow you to populate saved simulation waveform data into running Waveform Viewer instance.
xlSetUseHDL	Sets the 'Use behavioral HDL' option of blocks in a model of a Subsystem.
xlTBUtils	Provides programmatic access to several useful procedures such as layout, re-drawlines and getselected.

xilinx.analyzer

xilinx.analyzer is a MATLAB® class that provides an interface between the Model Composer model and Vivado® timing paths.

The Model Composer timing analysis is supported for all compilation targets. The **Perform analysis** drop down menu under the **Clocking** tab of the System Generator token provides two options for the trade-off between total runtime vs. accuracy of the Vivado timing data. If you select either the **Post Synthesis** or the **Post Implementation** option of **Perform analysis** and click the **Generate** button, then Vivado timing paths information is collected during the netlist generation. The xilinx.analyzer class is used to access Vivado timing paths information. The xilinx.analyzer class object processes Vivado timing paths to find 50 unique paths with the worst slack value. The unique timing paths are sorted in increasing value of slack and saved in the analyzer object.

The cross-probing between Vivado timing paths and the Model Composer model is made possible using the following API functions in the xilinx.analyzer class.

Table 119: **xilinx.analyzer Class Functions**

Function Name	Description	Function Argument
xilinx.analyzer	This is a constructor of the class. A call to the xilinx.analyzer constructor returns object of the class.	First argument is Model Composer model name. Second argument is path to already generated netlist directory.
isValid	Indicates if timing analysis data is valid or not. Use this API to make sure that the xilinx.analyzer class construction was successful.	No argument
getErrorMessage	Returns an error message string if the call to the class constructor or other API function had an error.	No argument
getStatus	Returns 'FAILED' if any of the timing paths in the model have a violation, i.e., negative slack.	No argument
getVivadoStage	Returns either Post Synthesis or Post Implementation. This is the Vivado design stage after which timing analysis was performed.	No argument
paths	Returns an array of MATLAB structures. Each structure contains data for a timing path.	A string that is equal to either 'setup' or 'hold'
violations	Returns an array of MATLAB structures. Each member of the array is a path structure with a timing violation.	A string that is equal to either 'setup' or 'hold'
print	Prints timing path information such as Slack, Path Delay, Levels of Logic, Name of Source and Destination blocks, and Source and Destination clocks.	An array of MATLAB structures for timing path data. The array can have one or more structures.
highlight	In the Model Composer model, highlights blocks for the timing path passed in the argument. Blocks that are already highlighted in the model will remain highlighted.	MATLAB structure for one timing path
highlightOnePath	In the Model Composer model, highlights blocks for the timing path passed in the argument. Before highlighting blocks for this path, the blocks that are already highlighted in the model will be unhighlighted.	MATLAB® structure for one timing path
unhighlight	In the Simulink® model, unhighlights all blocks currently highlighted.	No argument
disp	Displays a summary of timing analysis results on the MATLAB console, including the worst slack value among all timing paths.	No argument
delete	This is a destructor of the xilinx.analyzer class	No argument

Table 120: **Timing Path Data in a MATLAB Structure**

Field Name	Description
Slack	The double value containing timing slack for the path
Delay	Total Data Path delay for the path
Levels_of_Logic	Number of elements in Vivado design for the timing path. The number of HDL blocks in the timing path may be different from Levels_of_Logic.
Source	First HDL block in the timing path
Destination	Last HDL block in the timing path

Table 120: **Timing Path Data in a MATLAB Structure** *(cont'd)*

Field Name	Description
Source_Clock	Name of the clock domain for the source block
Destination_Clock	Name of the clock domain for the destination block
Path_Constraints	Timing constraint used for the path. For a multi-clock design, the path constraint can be a multi-clock timing constraint.
Block_Masks	Cell array where each element contains mask information for a HDL block.
Simulink_Names	Cell array where each element contains hierarchical name of a block in Model Composer model
Vivado_Names	Cell array where each element contains name of HDL block in Vivado database
Type	A timing violation type. The value is either 'setup' or 'hold'.

xilinx.analyzer - Construct xilinx.analyzer class object

Syntax

```
analyzer_object = xilinx.analyzer(<name_of_the_model>',
'<path_to_netlist_directory>')
```

- **Description:**

  A call to xilinx.analyzer constructor returns object of the class.

  The first argument is the name of the Model Composer model. The model must be open before the class constructor is called.

  The second argument is an absolute or relative path to the netlist directory. You must have read permission to the netlist directory.

  To access API functions of the xilinx.analyzer class use the object of the class as described below. To get more details for a specific API function type the following at the MATLAB command prompt:

  ```
  help xilinx.analyzer.<API_function>
  ```

- **Example:**

  ```
  //Construct class. Must give the model name and absolute or relative path
  to the
  //target directory

  >> timing_object = xilinx.analyzer('fixed_point_IIR', './
  netlist_for_timing_analysis')

  timing_object =

  Number of setup paths = 9
  Worst case setup slack = -1.6430
  ```

isValid – Check validity of Vivado timing paths

- **Syntax:**

```
result = analyzer_object.isValid();
```

- **Description:**

  If timing analysis data is valid then the result equals '1', otherwise it is '0'. Use this API to make sure that the xilinx.analyzer class construction was successful and the timing data was valid.

- **Example:**

```
//Determine if timing analysis data is valid

>> valid_status = timing_object.isValid()

valid_status =

     1
```

getErrorMessage - Get an error message

- **Syntax:**

```
result = analyzer_object.getErrorMessage();
```

- **Description:**

  Returns an error message string if the call to the class constructor or other API function had an error.

- **Example:**

```
//Determine if there was an error in the xilinx.analyzer constructor
//or in any of the API functions

>> err_msg = timing_object.getErrorMessage()

err_msg =

     ''
```

getStatus - Timing analysis status

- **Syntax:**

```
string = analyzer_object.getStatus();
```

- **Description:**

  The returned string is either 'PASSED' or 'FAILED'. If any of the timing paths have a violation, i.e. negative slack, then the timing analysis status is considered failed.

- **Example:**

```
//Determine if there were timing path violations in Simulink model

>> analysis_status = timing_object.getStatus()

analysis_status =

FAILED
```

getVivadoStage - Get Vivado design stage for timing analysis

- **Syntax:**

```
string = analyzer_object.getVivadoStage();
```

- **Description:**

  The returned string is the Vivado design stage after which timing analysis was performed and data collected in Vivado. The value is either 'Post Synthesis' or 'Post Implementation'.

- **Example:**

```
//Determine Vivado stage when timing data was collected

>> design_stage = timing_object.getVivadoStage()

design_stage =

Post Synthesis
```

paths - Access all timing paths

- **Syntax:**

```
<array_of_timing_paths_structure> =
analyzer_object.paths('<violation_type>');
```

- **Description:**

  The returned value is an array of MATLAB structures. Each structure contains data for a timing path, sorted in decreasing order of timing violation, i.e. in increasing order of slack value.

The argument 'violation_type' is either 'setup' or 'hold' string.

- **Example:**

```
//Return an array of the timing path structures

>> all_timing_paths = timing_object.paths('setup')

all_timing_paths =

1x9 struct array with fields:

    Slack
    Delay
    Levels_of_Logic
    Source
    Destination
    Source_Clock
    Destination_Clock
    Path_Constraints
    Block_Masks
    Simulink_Names
    Vivado_Names
    Type
```

***Note*:**

There are a total of nine timing paths in this timing analysis.

You can find the data fields in each timing path as shown in Example 1 in Additional Information.

violations - Access paths with timing violations

- **Syntax:**

```
<array_of_timing_paths_structure> =
analyzer_object.violations('<violation_type>');
```

- **Description:**

The returned value is an array of MATLAB structures. Each member of the array is data for a path with a timing violation. The array elements are sorted in decreasing order of timing violation. If there are no timing violations in the design then the API function returns an empty array.

The argument 'violation_type' is either 'setup' or 'hold'.

- **Example:**

```
//Return an array of timing paths with setup violations

>> violating_paths = timing_object.violations('setup')

violating_paths =

1x2 struct array with fields:

    Slack
    Delay
    Levels_of_Logic
    Source
    Destination
    Source_Clock
    Destination_Clock
    Path_Constraints
    Block_Masks
    Simulink_Names
    Vivado_Names
    Type
```

There are a total of two paths with violations in this timing analysis.

You can find the data fields in each timing path as shown in Example 3 in Additional Information.

print - Print timing path information

- **Syntax:**

```
analyzer_object.print(<timing_path_structures>);
```

- **Description:**

Prints timing data such as Slack, Path Delay, Levels of Logic, Name of Source and Destination blocks, Source and Destination clocks, Path Constraints, etc. for the input timing path structure.

The argument is an array of MATLAB structures with one or more elements.

- **Examples:**

```
//Print timing path information for path #1

>> timing_object.print(all_timing_paths(1))
Path Num            Slack (ns)             Delay (ns)            Levels
of Logic
Source/Destination Blocks            Source Clock      Destination
Clock               Path
Constraints
   1                             -1.6430
11.5690                     6
```

```
fixed_point_IIR/Delay1
clk                           clk
create_clock -name clk -period 2 [get_ports clk]


fixed_point_IIR/IIR Filter Subsystem/Delay4

ans =

     1

//Print timing path information for path #3

>> timing_object.print(all_timing_paths(3))
Path Num            Slack (ns)             Delay (ns)             Levels
of Logic
Source/Destination Blocks            Source Clock      Destination
Clock               Path
Constraints
   1                               1.1320
0.5270                         0
fixed_point_IIR/Delay1
clk                              clk
create_clock -name clk -period 2 [get_ports clk]


fixed_point_IIR/Delay1

ans =

     1

//Print timing path information for path #2 from violating_paths array

>> timing_obj.print(violating_paths(2))
Path Num            Slack (ns)             Delay (ns)             Levels
of Logic
Source/Destination Blocks            Source Clock      Destination
Clock               Path
Constraints
   1                               -1.3260
11.2520                       6
fixed_point_IIR/Delay1
clk                           clk
create_clock -name clk -period 2 [get_ports clk]


fixed_point_IIR/Delay2

ans =

     1
```

highlight - Highlight design blocks for a timing path

- **Syntax:**

```
analyzer_object.highlight(<timing_path_structure>);
```

- **Description:**

This API highlights HDL blocks for the timing path passed in the argument. It doesn't change the highlighting of a block from other paths, so more than one timing path can be highlighted if you use this function repeatedly.

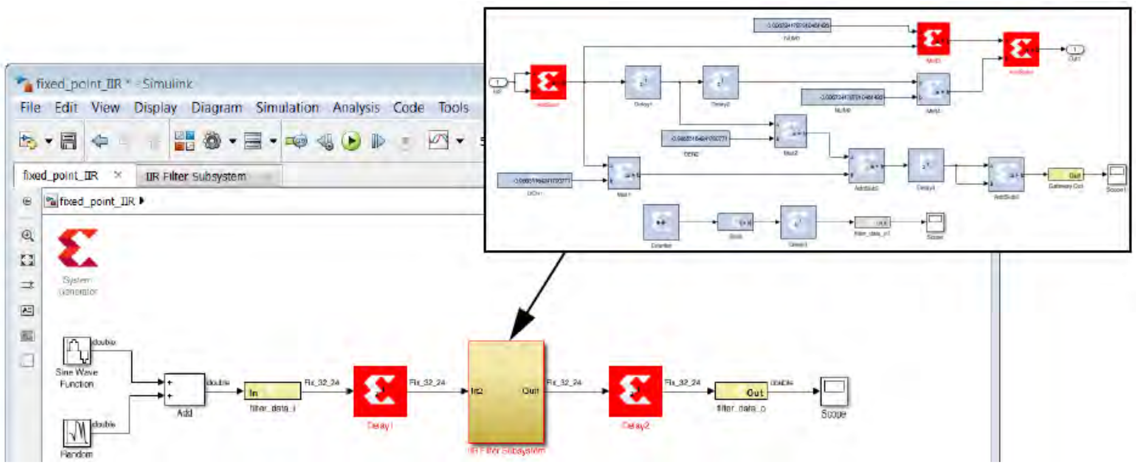
The argument is the MATLAB structure for one timing path.

- **Example:**

```
//Highlight Simulink model blocks in the selected path
//Don't change highlighting of currently highlighted blocks in the model

>> [result, err_msg] = timing_object.highlight(all_timing_paths(1));
```

Highlighted Model Composer model blocks appear as shown below.

Figure 420: **HDL Model Blocks**



highlightOnePath - Highlight design blocks for one timing path

- **Syntax:**

```
analyzer_object.highlightOnePath(<timing_path_structure>);
```

- **Description:**

  This API highlights HDL blocks for the timing path passed in the argument. If a block from other paths is already highlighted then it will be unhighlighted first, so only one path is highlighted at a time.

  The argument is the MATLAB structure for one timing path.

- **Example:**

```
//Highlight a single path in Model Composer model, and unhighlight
currently
//highlighted paths

>> [result, err_msg] = timing_object.highlightOnePath(violating_paths(2));
```

unhighlight - Unhighlight design blocks

- **Syntax:**

```
analyzer_object.unhighlight();
```

- **Description:**

  This API unhighlights blocks that are already highlighted. The blocks in Model Composer model are displayed in their original colors.

- **Example:**

```
//Unhighlight any Simulink block that is currenly highlighted

>> [result, err_msg] = timing_object.unhighlight();
```

disp - Display summary of timing analysis

- **Syntax:**

```
analyzer_object.disp();
```

- **Description:**

  This API displays the summary of timing paths on the MATLAB console, including the worst slack value.

- **Example:**

```
//Display a summary of timing analysis

>> timing_object.disp()
Number of setup paths = 9
Worst case setup slack = -1.6430
```

delete - Delete xilinx.analyzer class object

- **Syntax:**

```
analyzer_object.delete();
```

- **Description:**

  This is a destructor for the xilinx.analyzer class.

- **Example:**

```
//Delete xilinx.analyzer object, i.e., timing_object

>> timing_object.delete();
```

Additional Information

Accessing data fields of timing path structures:

- **Example 1: Data for timing path #1:**

```
//Return the data fields for the timing path with the worst slack

>> all_timing_paths(1)

ans =

                Slack: -1.6430
                Delay: 11.5690
      Levels_of_Logic: 6
               Source: 'fixed_point_IIR/Delay1'
          Destination: 'fixed_point_IIR/IIR Filter Subsystem/Delay4'
         Source_Clock: 'clk'
    Destination_Clock: 'clk'
     Path_Constraints: 'create_clock -name clk -period 2 [get_ports ...'
          Block_Masks: {1x5 cell}
       Simulink_Names: {1x5 cell}
         Vivado_Names: {1x5 cell}
                 Type: 'setup'
```

- **Example 2: Data for timing path #3:**

```
//Return the data fields for a timing path

>> all_timing_paths(3)

ans =

                Slack: 1.1320
                Delay: 0.5270
      Levels_of_Logic: 0
               Source: 'fixed_point_IIR/Delay1'
```

```
          Destination: 'fixed_point_IIR/Delay1'
         Source_Clock: 'clk'
    Destination_Clock: 'clk'
     Path_Constraints: 'create_clock -name clk -period 2 [get_ports ...'
          Block_Masks: {'fprintf('','COMMENT: begin icon graphics')...'}
       Simulink_Names: {'fixed_point_IIR/Delay1'}
         Vivado_Names: {'fixed_point_iir.fixed_point_iir_struct.delay1'}
```

Type: 'setup'

- **Example 3: Data for path #1 in violating_paths array:**

```
//Return the data fields in a timing path with timing violations

>> violating_paths(1)

ans =

                Slack: -1.6430
                Delay: 11.5690
      Levels_of_Logic: 6
               Source: 'fixed_point_IIR/Delay1'
          Destination: 'fixed_point_IIR/IIR Filter Subsystem/Delay4'
         Source_Clock: 'clk'
    Destination_Clock: 'clk'
     Path_Constraints: 'create_clock -name clk -period 2 [get_ports ...'
          Block_Masks: {1x5 cell}
       Simulink_Names: {1x5 cell}
         Vivado_Names: {1x5 cell}
                 Type: 'setup'
```

xilinx.environment.getcachepath and xilinx.environment.setcachepath

`xilinx.environment.getcachepath` is used to get the path Model Composer currently uses to store the simulation cache.

`xilinx.environment.setcachepath` is used to change the path Model Composer uses to store the simulation cache.

Syntax

```
xilinx.environment.getcachepath

xilinx.environment.setcachepath(path)
```

Description

When you simulate a Simulink model containing Xilinx IP in Model Composer, the Vivado simulator simulation data for that particular IP configuration is cached to speed up the simulation.

Model Composer establishes the simulation cache at a default location at startup, and you can determine the current path to the simulation cache with the `xilinx.environment.getcachepath` command. If you need to change the location of the simulation cache, use the `xilinx.environment.setcachepath` command. You will need to have write permission on the destination path directory. The new path will apply for the remainder of your Model Composer session.

One reason you would use `xilinx.environment.setcachepath` is to set the path if you do not have write permission on the default directory Model Composer uses for caching simulation data.

Examples

Example 1: Getting the current simulation cache path.

```
>> xilinx.environment.getcachepath

ans =

C:\Users\my_login/AppData/Local/Xilinx/Sysgen/SysgenVivado/win64.o
```

Example 2: Setting a new simulation cache path.

```
>> xilinx.environment.setcachepath('C:\sim_cache')

ans =

C:\sim_cache

>> xilinx.environment.getcachepath

ans =

C:\sim_cache
```

xilinx.resource_analyzer

xilinx.resource_analyzer is a MATLAB class that enables cross-probing between the Model Composer model and Vivado resource utilization data.

The Model Composer resource analysis is supported for all compilation targets. The **Perform analysis** drop down menu under the **Clocking** tab of the System Generator token provides two options for the trade-off between Vivado tools runtime vs. accuracy of the resource utilization data. If you select either the **Post Synthesis** or the **Post Implementation** option of **Perform analysis** and click the **Generate** button, then Vivado resource utilization information is collected during the netlist generation. Once the netlist generation has completed, the xilinx.resource_analyzer class is used to access this Vivado resource utilization results. The xilinx.resource_analyzer class object processes Vivado resource utilization data to display the number of resources (BRAMs, DSPs, Registers, and LUTs) used in the Simulink model, as well as by the subsystems and low-level blocks in the model.

The cross-probing between Vivado resource utilization results and the Model Composer model is made possible through the following API functions in the xilinx.resource_analyzer class.

Table 121: **Functions in xilinx.resource_analyzer Class**

Function Name	Description	Function Argument
xilinx.resource_analyzer	This is a constructor of the class. A call to the xilinx.resource_analyzer constructor returns object of the class.	First argument is design model name. Second argument is path to already generated netlist directory.
getVivadoStage	Returns either `Post Synthesis` or `Post Implementation`. This is the Vivado design stage after which resource analysis was performed.	No argument
getDevicePart	Returns a string for device part, package and speed grade for the device in which the design will be implemented.	No argument
getDeviceResource	Returns a string for the total count of the specified type of resource in the target Xilinx device.	(Optional) Resource type.Resource types are: BRAMs, DSPs, Registers, or LUTs.
printDeviceResources	Prints the total number of BRAMs, DSPs, Registers, and LUTs available on the target Xilinx device. The counts are printed in the MATLAB console.	No argument.
getCount	Returns a count for the particular resource type used by a block or subsystem.	(Optional) First argument is a Simulink handle or pathname for the block. (Optional) Second argument is Resource type. Resource types are: BRAMs, DSPs, Registers, or LUTs.
print	Returns a count for the particular resource type used by a block or subsystem.	(Optional) First argument is a Simulink handle or pathname for the block or subsystem. (Optional) Second argument is resource type. Resource types are: BRAMs, DSPs, Registers, or LUTs.
getDistribution	Returns three values: An array of MATLAB structures. Each element in the array is a structure containing the name of a block or subsystem directly under the subsystem in the argument, with a key-value pair of the resource type and number of resources used by that sub block or subsystem. A count of the resources used by the self (the block or subsystem specified in the argument). A count of the resources used by both blocks and subsystems combined.	First argument is a Simulink handle or pathname for the block or subsystem. Second argument is resource type. Resource types are: BRAMs, DSPs, Registers, or LUTs.

Table 121: **Functions in xilinx.resource_analyzer Class** *(cont'd)*

Function Name	Description	Function Argument
getErrorMessage	Returns an error message string if the call to the class constructor or other API function had an error.	No argument
highlight	In the Simulink model, highlights the specified block or subsystem with yellow color and red border.	A Simulink handle or pathname for the block to highlight.
unhighlight	In the Simulink model, unhighlights a block which is currently highlighted.	(Optional) A Simulink handle or pathname for the block to unhighlight.
delete	This is a destructor of the xilinx.resource_analyzer class	No argument

Table 122: **Resource Data in a MATLAB Structure**

Field Name	Description
BRAMs	Count of block RAM resources for a block or subsystem. BRAMs are counted in this way: • RAMB36E: 1 BRAM • RAMB36E: 1 BRAM • RAMB18E: 0.5 BRAM • FIFO18E: 0.5 BRAM Variations of Primitives (for example, RAM36E1 and RAM36E2) are all counted in the same way. Total BRAMs = (Number of RAMB36E) + (Number of FIFO36E) + 0.5 (Number of RAMB18E + Number of FIFO18E)
DSPs	Count of DSP48 resources utilized by a block or subsystem.
Registers	Count of Flip-Flops and Latches used by the design is reported as the number of Registers utilized by the design model, a particular block, or a subsystem.
LUTs	Count of all LUT type resources utilized by a block or subsystem.

xilinx.resource_analyzer – Construct xilinx.resource_analyzer class object

Syntax

```
resource_analyzer_obj =
xilinx.resource_analyzer('<name_of_the_model>','<path_to_netlist_directory>'
);
```

Description

A call to xilinx.resource_analyzer constructor returns object of the class.

The first argument is the name of the Model Composer model. The model must be open before the class constructor is called.

The second argument is an absolute or relative path to the netlist directory. You must have read permission to the netlist directory.

To access API functions of the xilinx.resource_analyzer class use the object of the class as described below. To get more details for a specific API function type the following at the MATLAB command prompt:

```
help xilinx.resource_analyzer.<API_function>
```

Example

```
//Construct class. Must give the model name and absolute or relative path
to the
//target directory

>> res_obj = xilinx.resource_analyzer('test_decimator', './
netlist_for_resource_analysis')

res_obj =

Resources used by: test_decimator
BRAMs => 0.5
DSPs => 1
Registers => 273
LUTs => 153
```

getVivadoStage – Get Vivado design stage for resource analysis

Syntax

```
string = resource_analyzer_obj.getVivadoStage();
```

Description

The returned string is the Vivado design stage after which resource analysis was performed and data collected in Vivado. The value is either `Post Synthesis` or `Post Implementation`.

Example

```
//Determine Vivado stage when resource data was collected

>> design_stage = res_obj.getVivadoStage()

design_stage =

Post Synthesis
```

getDevicePart – Get target Xilinx device part name

Syntax

```
string = resource_analyzer_obj.getDevicePart();
```

Description

Gets the name of the Xilinx device to which your design is targeted.

Example

```
//Get the Xilinx part in which you will implement your design

>> part_name = res_obj.getDevicePart()

part_name =

xc7k325tfbg676-3
```

getDeviceResource – Get number of resources in target device

Syntax

```
total_resource_count =
resource_analyzer_obj.getDeviceResource(<resource_type>);
```

Description

The returned value is the total number of a particular type of resource contained in the Xilinx device for which you are targeting your design.

The `resource_type` may be:

- BRAMs - Block RAM and FIFO primitives
- DSPs - DSP48 primitives
- Registers - Registers and Flip-Flops
- LUTs - All LUT types combined

If no `resource_type` is provided, the command returns a MATLAB structure containing all device resources.

Example

```
//Determine the total number of Block RAMs in the Xilinx device

>> total_brams = res_obj.getDeviceResource('BRAMs')
```

```
total_brams =

445

//Determine the total number of Block RAMs, DSP blocks, Registers, and LUTs
in the
//Xilinx device

>> total_resource_count = res_obj.getDeviceResource

total_resource_count =

    BRAMs: '445'
     DSPs: '840'
Registers: '407600'
     LUTs: '203800'
```

printDeviceResources – Print number of resources in target device

Syntax

```
resource_analyzer_obj.printDeviceResources();
```

Description

Prints the number of all types of resources in the used Xilinx device. The output is printed in the MATLAB console.

Examples

```
//Print the number of all types of resources contained in the target Xilinx
device

>> res_obj.printDeviceResources()

BRAMs => 445
DSPs => 840
Registers => 407600
LUTs => 203800
```

getCount – Get resource utilization for subsystem or block

Syntax

```
<block_resource_count> =
resource_analyzer_obj.getCount(<blockID>,<resource_type>);
```

Description

The returned value is the total number of a particular type of resource used in the specified subsystem or block.

The `blockID` can be either a Simulink handle or a pathname (a hierarchical name) for the subsystem or block.

The `resource_type` may be:

- BRAMs - Block RAM and FIFO primitives
- DSPs - DSP48 primitives
- Registers - Registers and Flip-Flops
- LUTs - All LUT types combined

If no `resource_type` is provided, the command returns a MATLAB structure containing all device resources.

Example

```
// Return register resource utilization for Simulink block with pathname
// test_decimator/addr_gen

>> regs_in_block = res_obj.getCount('test_decimator/addr_gen', 'Registers')

ans =

         105
//Return resource utilization for the entire Simulink model

>> total_resource_count = res_obj.getCount()

Resources used by: test_decimator
BRAMs => 0.5
DSPs => 1
Registers => 273
LUTs => 153
```

print – Prints all resources used by a subsystem or block

Syntax

```
resource_analyzer_obj.print(<blockID>);
```

Description

Prints all resources (for all resource types: BRAMs, Registers, DSPs, and LUTs) used by a subsystem or block, in key-value pair. Resources are printed in the MATLAB console.

If you enter a `blockID` (which can be either a Simulink handle or a pathname), all resources used by the specified block or subsystem will be printed in the MATLAB console.

If no `blockID` argument is provided, all resources used by the top-level design will be printed in the MATLAB console.

Example

```
// Print resource utilization for Simulink subsystem with pathname
// test_decimator/addr_gen

>> res_obj.print('test_decimator/subsystem1')
Resources used by: test_decimator/subsystem1
BRAMs => 0.5
DSPs => 1
Registers => 49
LUTs => 97

//Print resource utilization for the entire Simulink model

>> res_obj.print()
Resources used by: test_decimator
BRAMs => 0.5
DSPs => 1
Registers => 273
LUTs => 153
```

getDistribution – Get count of each resource type used by each block and subsystem under a specified subsystem

Syntax

```
[<distribution_array>, <self_count>, <total_count>] =
resource_analyzer_obj.getDistribution(<blockId>, <resource_type>)
```

Description

Returns count for the specified type of resource used by each block and subsystem directly under the subsystem passed as the argument.

The three returned values are:

- An array of MATLAB structures. Each element in the array is a structure containing the name of a block or subsystem directly under the subsystem in the argument, with a key-value pair of the resource type and number of resources used by that sub block or subsystem.
- A count of the resources used by the self (the block or subsystem specified in the argument).
- A count of the resources used by both blocks and subsystems combined.

The `blockID` can be either a Simulink handle or a pathname (a hierarchical name) for the subsystem or block. If no `blockID` is provided, then the command assumes the top-level module.

The `resource_type` may be:

- BRAMs - Block RAM and FIFO primitives

- DSPs - DSP48 primitives
- Registers - Registers and Flip-Flops
- LUTs - All LUT types combined

Example

```
// Return Register resource distribution for Simulink block with pathname
// test_decimator. This is top level of the design

>> [res_dist, self, total] = res_obj.getDistribution
('test_decimator','Registers')

res_dist =

1x8 struct array with fields:

    Name
    Hier_Name
    Count


self =

         119


total =

         273
//Return resource utilization for the entire Simulink model

>> total_resource_count = res_obj.getCount()

Resources used by: test_decimator
BRAMs => 0.5
DSPs => 1
Registers => 273
LUTs => 153
```

getErrorMessage – Get an error message

Syntax

```
result = resource_analyzer_obj.getErrorMessage();
```

Description

Returns an error message string if the call to the class constructor or other API function had an error.

Example

```
//Determine if there was an error in the xilinx.resource_analyzer
constructor
//or in any of the API functions

>> err_msg = res_obj.getErrorMessage()

err_msg =

     ''
```

highlight – Highlight design subsystems and blocks

Syntax

```
resource_analyzer_obj.highlight(<blockId>)
```

Description

This API highlights blocks in the Simulink model. Highlighted blocks in the Model Composer model are displayed in yellow and outlined in red. Highlighting blocks using this command does not change the highlighting of other blocks currently highlighted, so more than one block can be highlighted if you use this function repeatedly.

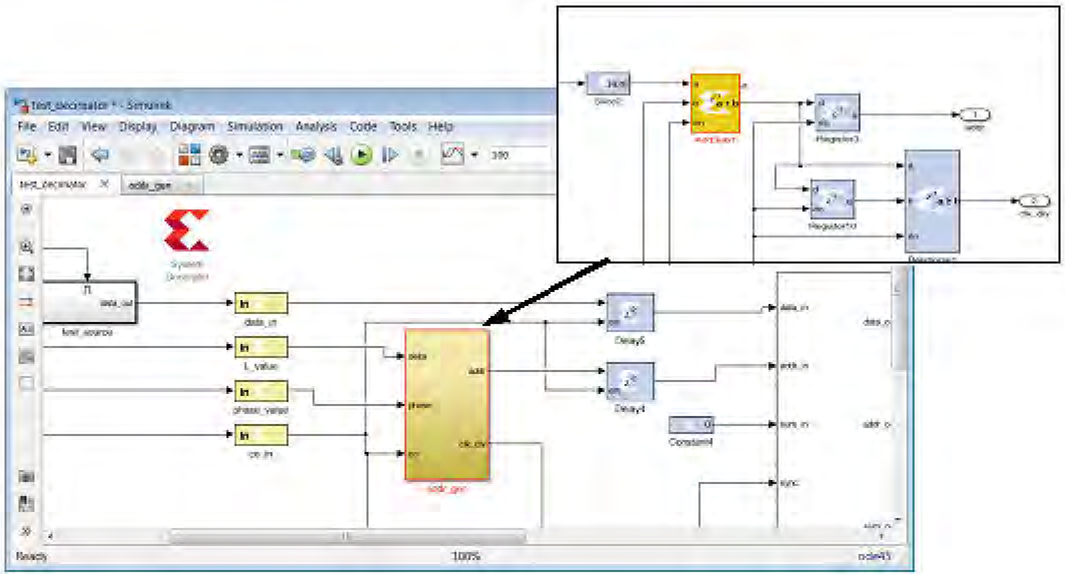
When you enter a `blockID` (which can be either a Simulink handle or a pathname) for a block or subsystem, the specified block or subsystem will be highlighted in the Simulink model. When the block/subsystem is highlighted then all parent subsystems up to the top level are also highlighted. When the top level module handle is provided as the `highlight` function argument no block is highlighted, but the Simulink model display changes to the top level, showing all blocks and subsystems at the top level.

Example

```
//Highlight Simulink block with pathname fixed_point_IIR/IIR Filter
Subsystem/Mult1

>> res_obj.highlight('test_decimator/addr_gen/AddSub1')
```

Highlighted Model Composer model blocks appear as shown below.

Figure 452: **HDL Model Blocks**



unhighlight – Unhighlight design subsystems and blocks

Syntax

```
resource_analyzer_obj.unhighlight(<blockId>)
```

Description

This API unhighlights blocks that are currently highlighted in the Simulink model. When they are unhighlighted, the blocks in the Model Composer model are displayed in their original colors.

If you enter a `blockID` (which can be either a Simulink handle or a pathname) for a block or subsystem, the specified block or subsystem will be unhighlighted in the Simulink model. When the block/subsystem is unhighlighted, all parent subsystems up to the top level are also unhighlighted.

If no `blockID` argument is provided, all currently highlighted blocks and subsystems will be unhighlighted.

Example

```
//Unhighlight Simulink block with pathname test_decimator/addr_gen/Register4
>> res_obj.unhighlight('test_decimator/addr_gen/Register4')

//Unhighlight all Simulink blocks that are currenly highlighted
>> res_obj.unhighlight();
```

delete – Delete xilinx.resource_analyzer class object

Syntax

```
resource_analyzer_obj.delete();
```

Description

This is a destructor for the xilinx.resource_analyzer class.

Example

```
//Delete xilinx.resource_analyzer object, i.e., res_obj

>> delete(res_obj);
```

OR

```
>> res_obj.delete();
```

xilinx.utilities.importBD

`xilinx.utilities.importBD` imports a platform framework created in the Vivado IP integrator into a Model Composer model. The command provides an accelerated way to enter the Model Composer circuitry into the design. xilinx.utilities.importBD parses the platform framework for potential Model Composer ports and interfaces and creates a sample stub in the Simulink model.

Inputs to the xilinx.utilities.importBD command are the Vivado project to be imported and the name of the model to be created in Model Composer.

Syntax

```
xilinx.utilities.importBD(vivado_project,matlab_file)
```

Description

xilinx.utilities.importBD parses the platform framework Vivado project for potential Model Composer ports and interfaces and creates a sample stub to speed the development of the Model Composer model.

```
xilinx.utilities.importBD('<path_to_vivado_project_directory>/
<project_name>.xpr',
'mynewmodel')

xilinx.utilities.importBD('C:\test_impportBD\platform.xpr', 'mynewmodel')
```

xlAddTerms

xlAddTerms is similar to the addterms command in Simulink, in that it adds blocks to terminate or drive unconnected ports in a model. With xlAddTerms, output ports are terminated with a Simulink terminator block, and input ports are correctly driven with either a Simulink or HDL constant block. Additionally HDL gateway blocks can also be conditionally added.

The optionStruct argument can be configured to instruct xlAddTerms to set a block's property (e.g. set a constant block's value to 5) or to use different source or terminator blocks.

Syntax

```
xlAddTerms(arg1,optionStruct)
```

Description

In the following description, 'source block' refers to the block that is used to drive an unconnected port. And 'term block' refers to the block that is used to terminate an unconnected port.

```
xlAddTerms(arg1,optionStruct)
```

xlAddTerms takes either 1 or 2 arguments. The second argument, optionStruct argument is optional. The first argument can be the name of a system, or a block list.

Table 123: **xlAddTerms Arguments**

arg1	Description
gcs	A string-handle of the current system
'top/test1'	A string-handle of a system called test1. In this case, xlAddTerms is passed a handle to a system. This will run xlAddTerms on all the blocks under test1, including all children blocks of Subsystems.
{'top/test1'}	A block list of string handles. In this case, xlAddTerms is passed a handle to a block. This will run xlAddTerms only on the block called test1, and will not process child blocks.
{'t/b1';'t/b2';'t/b3'}	A block list of string handles.
[1;2;3]	A block list of numeric handles.

The optionStruct argument is optional, but when included, should be a MATLAB structure. The following table describes the possible values in the structure. The structure field names (as is true with all MATLAB structure field names) are case sensitive.

Table 124: **optionStruct Arguments**

optionStruct	Description

Source	xlAddTerms can terminate in-ports using any source block (refer to SourceWith field). The parameters of the source block can be specified using the Source field of the optionStruct by passing the parameters as sub-fields of the Source field. The Source field prompts xlAddTerms to do a series of set_params on the source block. Since it is possible to change the type of the source block, it is left to the user to ensure that the parameters here are relevant to the source block in use.
	E.g. when a Simulink constant block is used as a Source Block, setting the block's value to 10 can be done with:
	`Source.value = '10'`
	And when a HDL Constant block is used as a Source Block, setting the constant block to have a value of 10 and of type UFIX_32_0 can be done with:
	`Source.const = '10';`
	`Source.arith_type='Unsigned';`
	`Source.bin_pt=0;`
	`Source.n_bits=32;`

SourceWith	The SourceWith field allows the source block to be specified. Default is to use a constant block. SourceWith has two sub-fields which must be specified. SourceWithBlock: A string specifying the full path and name of the block to be used. e.g. 'built-in/Constant' or 'xbsIndex_r3/AddSub'. SourceWithPort: A string specifying the port number used to connect. E.g. '1' or '3' Specifying '1' instructs xlAddTerms to connect using port 1, etc.
TermWith	The TermWith Field allows the term block to be specified. Default is to use a Simulink terminator block. TermWith has two sub-fields which must be specified. TermWithBlock: A string specifying the full path and name of the block to be used. e.g. 'built-in/Terminator' or 'xbsIndex_r3/AddSub'. TermWithPort: A string specifying the port number used to connect. E.g. '1' or '3' Specifying '1' instructs xlAddTerms to connect using port 1, etc.
UseGatewayIns	Instructs xlAddTerms to insert HDL gateway ins when required. The existence of the field is used to denote insertion of gateway ins. This field must not be present if gateway ins are not to be used.

GatewayIn	If gateway ins are inserted, their parameters can be set using this field, in a similar way as for Source and Term.
	For example,
	`GatewayIn.arith_type='Unsigned';`
	`GatewayIn.n_bits='32'`
	`GatewayIn.bin_pt='0'`
	will set the gateway in to output a ufix_32_0.

UseGatewayOuts	Instructs xlAddTerms to insert HDL gateway outs when required. The existence of the field is used to denote insertion of gateway outs. This field must not be present if gateway outs are not to be used.

Table 124: **optionStruct Arguments** *(cont'd)*

optionStruct	Description

GatewayOut	If gateway outs are inserted, their parameters can be set using this field, in a similar way as for Source and Term.
	For example,
	`GatewayOut.arith_type='Unsigned';`
	`GatewayOut.n_bits='32'`
	`Gatewayout.bin_pt='0'`
	will set the gateway out to take an input of ufix_32_0.

RecurseSubsystems	Instructs xlAddTerm to recursively run xlAddTerm under all child Subsystems. Expects a scalar number, 1 or 0.

Examples

Example 1: Runs xlAddTerms on the current system, with the default parameters: constant source blocks are used, and gateways are not added. Subsystems are recursively terminated.

```
xlAddTerms(gcs);
```

Example 2: runs xlAddTerms on all the blocks in the Subsystem tt./mySubsystem.

```
xlAddTerms(find_system('tt/mySubsystem','SearchDepth',1));
```

Example 3: runs xlAddTerms on the current system, setting the source block's constant value to 1, using gateway outs and changing the term block to use a Simulink display block.

```
s.Source.const = '10';
s.UseGatewayOuts = 1;
s.TermWith.Block = 'built-in/Display';
s.TermWith.Port = '1';
s.RecurseSubsystem = 1;
xlAddTerms(gcs,s);
```

Remarks

Note that field names are case sensitive. When using the fields 'Source', 'GatewayIn' and 'GatewayOut', users have to ensure that the parameter names to be set are valid.

See Also

xlTBUtils

xlConfigureSolver

The `xlConfigureSolver` function configures the Simulink solver settings of a model to provide optimal performance during Model Composer simulation.

Syntax

```
xlConfigureSolver(<model_handle>);
```

Description

The xlConfigureSolver function configures the model referred to by <model_handle>. <model_handle> canbe a string or numeric handle to a Simulink model. Library models are not supported by this function since they have no simulation solver parameters to configure.

For optimal performance during Model Composer simulation, the following Simulink simulation configuration parameters are set:

```
'SolverType' = 'Variable-step'
'Solver' = 'VariableStepDiscrete'
'SolverMode' = 'SingleTasking'
```

xlfda_denominator

The xlfda_denomiator function returns the denominator of the filter object stored in the Xilinx FDATool block.

Syntax

```
[den] = xlfda_denominator(FDATool_name);
```

Description

Returns the denominator of the filter object stored in the Xilinx FDATool block named FDATool_name, or throws an error if the named block does not exist. The block name can be local (e.g. 'FDATool'), relative (e.g. '../../FDATool'), or absolute (e.g. 'untitled/foo/bar/FDATool').

See Also

xlfda_numerator, FDATool

xlfda_numerator

The xlfda_numerator function returns the numerator of the filter object stored in the Xilinx FDATool block.

Syntax

```
[num] = xlfda_numerator(FDATool_name);
```

Description

Returns the numerator of the filter object stored in the Xilinx FDATool block named FDATool_name, or throws an error if the named block does not exist. The block name can be local (e.g. 'FDATool'), relative (e.g. '../../FDATool'), or absolute (e.g. 'untitled/foo/bar/FDATool').

See Also

xlfda_denominator, FDATool

xlGenerateButton

The xlGenerateButton function provides a programmatic way to invoke the Model Composer code generator.

Syntax

```
status = xlGenerateButton(sysgenblock)
```

Description

xlGenerateButton invokes the Model Composer code generator and returns a status code. Invoking xlGenerateButton with a HDL block as an argument is functionally equivalent to opening the System Generator token, and clicking on the **Generate** button. The following is list of possible status codes returned by xlGenerateButton.

Table 125: **xlGenerateButton Status Codes**

Status	Description
1	Canceled
2	Simulation running
3	Check param error
4	Compile/generate netlist error
5	Netlister error
6	Post netlister script error

Table 125: **xlGenerateButton Status Codes** *(cont'd)*

Status	Description
7	Post netlist error
8	Post generation error
9	External view mismatch when importing as a configurable Subsystem

See Also

xlgetparam and xlsetparam, xlgetparams, System Generator block

xlgetparam and xlsetparam

Used to get and set parameter values in the System Generator token. Both functions are similar to the Simulink get_param and set_param commands and should be used for the System Generator token instead of the Simulink functions.

Syntax

```
[value1, value2, ...] = xlgetparam(sysgenblock, param1, param2, ...)

xlsetparam(sysgenblock, param1, value1, param2, value2, ...)
```

Description

The System Generator token differs from other blocks in one significant manner; multiple sets of parameters are stored for an instance of a System Generator token. The different sets of parameters stored correspond to different compilation targets available to the System Generator token. The 'compilation' parameter is the switch used to toggle between different compilation targets stored in the System Generator token. In order to get or set parameters associated with a particular compilation type, it is necessary to first use xlsetparam to change the 'compilation' parameter to the correct compilation target, before getting or setting further values.

```
[value1, value2, ...] = xlgetparam(sysgenblock, param1, param2, ...)
```

The first input argument of xlgetparam should be a handle to the System Generator token block. Subsequent arguments are taken as names of parameters. The output returned is an array that matched the number of input parameters. If a requested parameter does not exist, the returned value of xlgetparam is empty. The xlgetparams function can be used to get all the parameters for the current compilation target.

```
xlsetparam(sysgenblock, param1, value1, param2, value2, ...)
```

The xlsetparam function also takes a handle to a System Generator token as the first argument. Subsequent arguments must be provided in pairs, the first should be the parameter name and the second the parameter value.

Specifying the Compilation Parameter

The 'compilation' parameter on the System Generator token captures the compilation type chosen; for example 'HDL Netlist' or 'IP Catalog'. As previously stated, when a compilation type is changed, the System Generator token will remember all the options chosen for that particular compilation type. For example, when 'HDL Netlist' is chosen, the corresponding target directory could be set to 'hdl_dir', but when 'IP Catalog' is chosen, the target directory could point to a different location, for example 'ip_cat_dir'. Changing the compilation type causes the System Generator token to recall previous options made for that compilation type. If the compilation type is selected for the first time, default values are use to populate the rest of the options on the System Generator token.

When using xlsetparam to set the compilation type of a System Generator token, be aware of the above behaviour, since the order in which parameters are set is important; be careful to first set a block's 'compilation' type before setting any other parameters.

When xlsetparam is used to set the 'compilation' parameter, it must be the only parameter that is being set on that command. For example. the form below is not permitted:

```
xlsetparam(sysgenblock,'compilation','HDL Netlist', 'synthesis_tool',
'Vivado synthesis')
```

Examples

Example 1: Changing the synthesis tool used for HDL netlist.

```
xlsetparam(sysgenblock, 'compilation', 'HDL Netlist');
xlsetparam(sysgenblock, 'synthesis_tool', 'Vivado synthesis')
```

The first xlsetparam is used to set the compilation target to 'HDL Netlist'. The second xlsetparam is used to change the synthesis tool used to 'Vivado synthesis'.

Example 2: Getting family and part information.

```
[fam,part]=xlgetparam(sysgenblock,'xilinxfamily','part')
fam =
Virtex2
part =
xc2v1000
```

See Also

xlGenerateButton, xlgetparams

xlgetparams

The xlgetparams command is used to get all parameter values in a System Generator token associated with the current compilation type. The xlgetparams command can be used in conjunction with the xlgetparam and xlsetparam commands to change or retrieve a System Generator token's parameters.

Syntax

```
paramstruct = xlgetparams(sysgenblock_handle);
```

To get the sysgenblock_handle, enter gbc or gcbh at the MATLAB command line.

```
paramstruct = xlgetparams('chip/ System Generator');
paramstruct = xlgetparams(gcb);
paramstruct = xlgetparams(gcbh);
```

Description

All the parameters available to a System Generator token block can be retrieved using the `xlgetparams` command. For more information regarding the parameters, please refer to the System Generator token documentation.

```
paramstruct = xlgetparams(sysgenblock);
```

The first input argument of xlgetparams should be a handle to the System Generator token. The function returns a MATLAB structure that lists the parameter value pairs.

The compilation_lut parameter is another structure that lists the other compilation types that are stored in this System Generator token. Using xlsetparam to set the compilation type allows the parameters associated with that compilation type to be visible to either xlgetparams or xlgetparam.

See Also

xlGenerateButton, xlgetparam and xlsetparam

xlGetReOrderedCoeff

The xlGetReOrderedCoeff function provides the re-ordered coefficient set of a FIR Compiler block.

Syntax

```
A = xlGetReOrderedCoeff(new_coeff_set, returnType, block_handle)
```

Description

Note: All three parameters of this function are required.

new_coeff_set

The new coefficient set that needs to be loaded into an existing FIR Compiler. Must be supplied to the function in the original order.

block_handle

This is the FIR Compiler block handle in the design. If a FIR Compiler block is selected, then this block_handle can be specified as gcbh.

returnType

This parameter specifies the re-ordered coefficient or just the reload order information format. This value can be specified as either 'coeff' or 'index'. A 'coeff' return type will modify the required coefficient set and provide the re-arranged coefficient set that can be directly supplied to the FIR compiler block. The 'index' return type provides only the coefficient address vector using the new_coeff_set that needs to be processed manually.

Examples

Example 1:

If A is a row vector of coefficients, then the coefficients sorted in the reload order can be obtained as follows:

```
reload_order_coefficients = xlGetReOrderedCoeff(A,'coeff', gcbh)
```

In this example, reload_order_coefficients specifies the order in which coefficients contained in A should be passed to the FIR Compiler through the reload channel.

Example 2:

This example shows how to use an input text file is generated.

```
reload_order_coefficients =
xlGetReOrderedCoeff(A,'coeff',reload_<version>.txt)
```

Alternatively, the reload address vector can be obtained,

```
reload_order_coefficients = A(xlGetReOrderedCoeff(A,'index',gcbh))
```

See Also

FIR Compiler 7.2 block

xlOpenWaveFormData

Allows you to populate saved simulation waveform data into a running Waveform Viewer instance.

Syntax

```
xlOpenWaveFormData('C:/wavedata/model_name.wdb')
```

How to Use

1. Make sure an instance of Waveform Viewer is opened in the current Model Composer session.
2. Locate the waveform data file (`model_name.wdb`) you would like to open.

   Note: Waveform data are saved under the `wavedata` directory.
3. Type xlOpenWaveFormData ('C:/wavedata/model_name.wdb') in the MATLAB console. Make sure you enter the absolute path of the waveform data file.
4. Observe the waveform data in Waveform Viewer.

See Also

For information on using the Waveform Viewer to develop and troubleshoot your design, see this link in the Vivado Design Suite User Guide: Logic Simulation (UG900).

xlSetUseHDL

This function sets the 'Use behavioral HDL' option of blocks in a model or Subsystem.

Syntax

```
xlSetUseHDL(system, mode)
```

Description

The model or system specified in the parameter system is set to either use cores or behavioral HDL, depending on the mode. Mode is a number, where 0 refers to using cores, and 1 refers to using behavioral HDL.

Examples

Example 1:

```
xlSetUseHDL(gcs,0)
```

This call sets the currently selected system to use cores.

xlTBUtils

The xlTBUtils command provides access to several features that include access to the layout, rerouting functions and to functions that return selected blocks and lines.

Syntax

```
xlTBUtils(function, args)
e.g.
xlTBUtils('Layout',struct('verbose',1,'autoroute',0))
xlTBUtils('Layout',optionStruct)
xlTBUtils('Redrawlines',struct('autoroute',0))
xlTBUtils('RedrawLines',optionStruct)
[lines,blks]=xlTBUtils('GetSelected',handle,'all')
```

Description

xlTBUtils(function [,args])

The function argument specifies the name of the function to execute. Further arguments (if required) can be tagged on as supplementary arguments to the function call. Note that the function argument string is not case sensitive. Possible values are enumerated below and explained further in the relevant subtopics.

Table 126: **Function Argument**

Function	Description
'Layout'	Runs the layout algorithm on a model to place and reroute lines on the model. Layout can be customized using the option structure that is detailed below.
'RedrawLines'	Runs the routing algorithm on a model to reroute lines on the model. RedrawLines can be customized using the option structure detailed below.
'GetSelected'	Returns MATLAB Simulink handles to blocks and lines that are selected on the system in focus

'xlTBUtils('Layout',optionStruct)

Automatically places and routes a Simulink model. optionStruct is a MATLAB struct data-type, that contains the parameters for Layout. The optionStruct argument is optional.

Layout expects circuits to be placed left to right. After placement, Layout uses Simulink to autoroute the wire connections. Simulink will route avoiding anything visible on screen, including block labels. Setting "ignore_labels" will 'allow' Simulink to route over labels – after which it is possible to manually move the labels to a more reasonable location. Note that field names are case sensitive.

Table 127: **optionStruct Argument**

Field Names	Description [Default values]
x_pitch, y_pitch	The gaps (pitch) between block (pixels). x_pitch specifies the amount of spacing to leave between blocks horizontally, and y_pitch specifies vertical spacing. [30].
x_start, y_start	Left (x_start) and top(y_start) margin spacing (pixels). The amount of spacing to leave on the left and top of a model. [10].
autoroute	Turns on Simulink auto-routing of lines. (1 | 0) [1]
ignore_labels	When auto-routing lines, Simulink will attempt to auto-route around text labels. Setting ignore_labels to 1 will minimize text label size during the routing process.
sys	Name of the system to layout. [gcs]
verbose	When set to 1, a wait bar is shown during the layout process.

xlTBUtils('RedrawLines',optionStruct)

The RedrawLines command will redraw all lines in a Simulink model. If there are lines selected, only selected lines are redrawn otherwise all lines are redrawn. If a branch is selected, the entire line is redrawn; main trunk and all other sub-branches.

Table 128: **RedrawLines Command**

Field Names	Description [Default values]
autoroute	Turns on Simulink auto-routing of lines. (1 | 0) [1]
sys	Name of the system to layout. [gcs]

xlTBUtils('GetSelected',handle,arg)

The GetSelected command returns handles to selected blocks and lines of the system in focus. The argument handle returns the Simulink model, in which the blocks and lines are selected. You can also provide the model name in place of the argument handle. The argument `arg` should be one of the string values described in the following table.

Table 129: **GetSelected Command**

GetSelected	Description [Default values]
'all'	Gets both selected lines and blocks.
'lines'	Gets only selected lines.
'blocks'	Gets only selected blocks.

The GetSelected command will return an array with two items, an array of a structure containing line information (lines) and an array of block handles (blks). If the 'lines' argument is used, blks is an empty array; similarly when the 'blocks' argument is used, lines is an empty array.

Examples

Example 1a: Performing Layouts

```
a.verbose = 1;
a.autoroute= 0;
xlTBUtils('Layout',a);
```

This will invoke the layout tool with verbose on and autoroute off.

Example 1b: Performing Layouts

```
xlTBUtils('Layout',struct('verbose',1,'autoroute',0));
```

This will also invoke the layout tool with verbose on and autoroute off.

Example 2: Redrawing lines

```
xlTBUtils('Redrawlines',struct('autoroute',0));
```

This will redraw the lines of the current system, with auto-routing off.

Example 3: Getting selected lines and blocks

```
xlTBUtils('GetSelected',handle,'all')
lines =

1x3 struct array with fields:
  Handle
  Name
  Parent
  SrcBlock
  SrcPort
  DstBlock
  DstPort
  Points
  Branch
```

```
blks =

1.0e+003 *

3.0320
3.0480
```

This will return all selected lines and blocks in the current system. In this case, 3 lines and 2 blocks were selected. The first line handle can be accessed using the command

```

lines(1).Handle

ans =

3.0740e+003
```

The handle to the first block can be accessed using the command

```

blks(1)
ans =
3.0320e+003
```

Remarks

The actions performed by Layout and RedrawLines will not be in the undo stack. Save a copy of the model before performing the actions, in order to revert to the original model.

See Also

xlAddTerms

Programmatic Access

Model Composer API for Programmatic Generation

Introduction

A script of Model Composer for programmatic generation (PG API script) is a MATLAB M-function file that builds a Model Composer Subsystem by instantiating and interconnecting `xBlock`, `xSignal`, `xInport`, and `xOutport` objects. It is a programmatic way of constructing Model Composer diagrams (for example, Subsystems). As is demonstrated below with examples, the top-level function of a Model Composer programmatic script is its entry point and must be invoked through an `xBlock` contructor. Upon constructor exit, MATLAB adds the corresponding Model Composer Subsystem to the corresponding model. If no model is opened, a new untitled model is created and the Model Composer Subsystem is inserted into it.

The xBlock constructor creates an `xBlock` object. The object can be created from a library block or it can be a Subsystem. An `xSignal` object corresponds to a wire that connects a source block to a target. An `xInport` object instantiates a Simulink Inport and an `xOutport` object instantiates a Simulink Outport

The API also has one helper function, `xlsub2script` which converts a Simulink diagram to a programmatic generation script.

The API works in three modes: *learning mode*, *production mode*, and *debugging mode*. The learning mode allows you to type in the commands without having a physical script file. It is very useful when you learn the API. In this mode, all blocks, ports and Subsystems are added into a Simulink model named `untiled`. Please remember to run xBlock without any argument or to close the untitled model before starting a new learning session. The production mode has an M-function file and is invoked through the xBlock constructor. You will have a Subsystem generated. The Subsystem can be either in the existing model or can be inserted in a new model. The debugging mode works the same as the production mode except that every time a new object is created or a new connection is established, the Simulink diagram is rerouted. It is very useful when you debug the script that you set some break points in the script or single step the script.

xBlock

The `xBlock` constructor creates an `xBlock` object. The object can be created from a library block or it can be a Subsystem. The `xBlock` constructor can be used in three ways:

- to add a leaf block to the current Subsystem,
- to add a Subsystem to the current Subsystem,
- to attach a top-level Subsystem to a model.

The `xBlock` takes four arguments and is invoked as follows.

```
block = xBlock(source, params, inports, outports);
```

If the source argument is a string, it is expected to be a library block name. If the source block is in the xbsIndex_r4 library or in the Simulink built-in library, you can use the block name without the library name. For example, calling `xBlock('AddSub', ...)` is equivalent to `xBlock('xbsIndex_r4/AddSub',...)`. For a source block that is not in the `xbsIndex_r4` library or built-in library, you need to use the full path, for example, `xBlock('xbsTest_r4/Assert Relation', ...)`. If the source argument is a function handle, it is interpreted as a PG API function. If it is a MATLAB struct, it is treated as a configuration struct to specify how to attach the top-level to a model.

The `params` argument sets up the parameters. It can be a cell array for position-based binding or a MATLAB struct for name-based binding. If the source parameter is a block in a library, this argument must be a cell array. If the source parameter is a function pointer, this argument must be a cell array.

The `inports` and `outports` arguments specify how Subsystem input and output ports are bound. The binding can be a cell array for position-based binding or a MATLAB struct for name-based binding. When specifying an inport/outport binding, an element of a cell array can be an `xSignal`, an `xInport`, or an `xOutport` object. If the port binding argument is a MATLAB struct, a field of the struct is a port name of the block, a value of the struct is the object that the port is bound to.

The two port binding arguments are optional. If the arguments are missing when constructing the `xBlock` object, the port binding can be specified through the `bindPort` method of an `xBlock` object. The `bindPort` method is invoked as follows:

```
block.bindPort(inports, outports)
```

where `inports` and `outports` arguments specify the input and output port binding. In this case, the object block is create by `xBlock` with only two arguments, the source and the parameter binding.

Other `xBlock` methods include the following.

- `names = block.getOutportNames` returns a cell array of outport names.
- `names = block.getInportNames` returns a cell array of inport names.
- `nin = block.getNumInports` returns the number of inports.
- `nout = block.getNumoutports` returns the number of outports.
- `insigs = block.getInSignals` returns a cell array of in coming signals.
- `outsigs = block.getOutSignals` returns a cell array of out going signals.

xInport

An `xInport` object represents a Subsystem input port.

The constructor

```
port = xInport(port_name)
```

creates an xInport object with name port_name,

```
[port1, port2, port3, ...] = xInport(name1, name2, name2, ...)
```

creates a list of input port with names, and

```
port = xInport
```

creates an input port with an automatically generated name.

An `xInport` object can be passed for port binding.

METHODS

```
outsigs = port.getOutSignals
```

returns a cell array of out going signals.

xOutport

An `xOutport` object represents a Subsystem output port.

The constructor

```
port = xOutport(port_name)
```

creates an `xOutport` object with name `port_name`,

```
[port1, port2, port3, ...] = xOutport(name1, name2, name2, ...)
```

creates a list of output port with names, and

```
port = xOutport
```

creates an output port with an automatically generated name.

An `xOutport` object can be passed for port binding.

METHODS

```
port.bind(obj)
```

connects the object to port, where port is an xOutport object and obj is an `xSignal` or `xInport` object.

```
insigs = port.getInSignals
```

returns a cell array of incoming signals.

xSignal

An `xSignal` represents a signal object that connects a source to targets.

The constructor

```
sig = xSignal(sig_name)
```

creates an `xSignal` object with name sig_name,

```
[sig1, sig2, sig3, ...] = xSignal(name1, name2, name2, ...)
```

creates a list of signals with names, and

```
sig = xSignal
```

creates an `xSignal` for which a name is automatically generated.

An `xSignal` object can be passed for port binding.

METHODS

```
sig.bind(obj)
```

connects the `obj to sig`, where `sig` is an `xSignal` object and `obj` is an `xSignal` or an `xInport` object.

```
src = sig.getSrc
```

returns a cell array of the source objects that are driving the `xSignal` object. The cell array can have at most one element. If the source is an input port, the source object is an xInport object. If the source is an output port of a block, the source object is a struct, having two fields block and port. The block field is an `xBlock` object and the port field is the port index.

```
dst = sig.getDst
```

returns a cell array of the destination objects that the `xSignal` object is driving. Each element can be either a struct or an `xOutport` object. It is defined same as the return value of the `getSrc` method.

xlsub2script

`xlsub2script` is a helper function that converts a Subsystem into the top level of a Sysgen script.

`xlsub2script(Subsystem)` converts the Subsystem into the top-level script. The argument can also be a model.

By default, the generated M-function file is named after the name of the Subsystem with white spaces replaced with underscores. Once the `xlsub2script` finishes, a help message will guide you how to use the generated script. The main purpose of this `xlsub2script` function is to make learning Sysgen Script easier. This is also a nice utility that allows you to construct a Subsystem using graphic means and then convert the Subsystem to a PG API M-function.

`xlsub2script(block)`, where block is a leaf block, prints out the `xBlock` call that creates the block.

The following are the limitations of `xlsub2script`.

- If the Subsystem has mask initialization code that contains function calls such as `gcb`, `set_param`, `get_param`, add_block, and so on, the function will error out and you must modify the mask initialization code to remove those Simulink calls.
- If there is an access to global variables inside the Subsystem, you need add corresponding mask parameters to the top Subsystem that you run the `xlsub2script`.
- If a block's link is broken, that block is skipped.

`xlsub2script` can also be invoked as the following:

```
xlsub2script(subsyste, options)
```

where `options` is a MATLAB struct. The `options` struct can have two fields: `forcewrite`, and `basevars`.

If `xlsub2script` is invoked for the same Subsystem the second time, `xlsub2script` will try to overwrite the existing M-function file. By default, `xlsub2script` will pop up a question dialog asking whether to overwrite the file or not. If the `forcewrite` field of the `options` argument is set to be true or 1, `xlsub2script` will overwrite the M-function file without asking.

Sometimes a Subsystem is depended on some variables in the MATLAB base workspace. In that case, when you run `xlsub2script`, you want `xlsub2script` to pick these base workspace variables and generate the proper code to handle base workspace variables. The `basevars` field of the `options` argument is for that purpose. If you want `xlsub2script` to pick up every variable in the base workspace, you need to set the `basevars` field to be '`all`'. If you want `xlsub2script` to selectively pick up some variables, you can set the `basevars` field to be a cell array of strings, where each string is a variable name.

The following are examples of calling `xlsub2script` with the options argument:

```
xlsub2script(Subsystem, struct('forcewrite', true));
xlsub2script(Subsystem, struct('forcewrite', true, 'basevars',

 'all'));
options.basevars = {'var1', 'var2', 'var3');
xlsub2script(Subsystem, options);
xlsub2script(Subsystem, struct('basevars', {{'var1', 'var2',

 'var3'}}));
```

In MATLAB, if the field of a struct is a cell array, when you call the struct() function call, you need the extra {}.

xBlockHelp

`xBlockHelp(<block_name>)` prints out the parameter names and the acceptable values for the corresponding parameters. When you execute `xBlockHelp` without a parameter, the available blocks in the `xbsIndex_r4` library are listed.

For example, when you execute the following in the MATLAB command line:

```
       xBlockHelp('AddSub')
```

You'll get the following table in the transcript:

```
'xbsIndex_r4/AddSub' Parameter Table

Parameter                Acceptable value         Type
============             ==================       ========
mode                     'Addition'               String
                         'Subtraction'
                         'Addition or Subtraction'
------------             ------------------       --------
use_carryin              'off'                    String
                         'on'
------------             ------------------       --------
use_carryout             'off'                    String
                         'on'
------------             ------------------       --------
en                       'off'                    String
                         'on'
------------             ------------------       --------
latency                  An Int value             Int
------------             ------------------       --------
precision                'Full'                   String
                         'User Defined'
------------             ------------------       --------
arith_type               'Signed  (2's comp)'     String
                         'Unsigned'
------------             ------------------       --------
n_bits                   An Int value             Int
------------             ------------------       --------
bin_pt                   An Int value             Int
```

```
------------         ------------------       --------
quantization         'Truncate'               String
                     'Round  (unbiased: +/- Inf)'
------------         ------------------       --------
overflow             'Wrap'                   String
                     'Saturate'
                     'Flag as error'
------------         ------------------       --------
use_behavioral_HDL   'off'                    String
                     'on'
------------         ------------------       --------
pipelined            'off'                    String
                     'on'
------------         ------------------       --------
use_rpm              'off'                    String
                     'on'
------------         ------------------       --------
```

PG API Examples

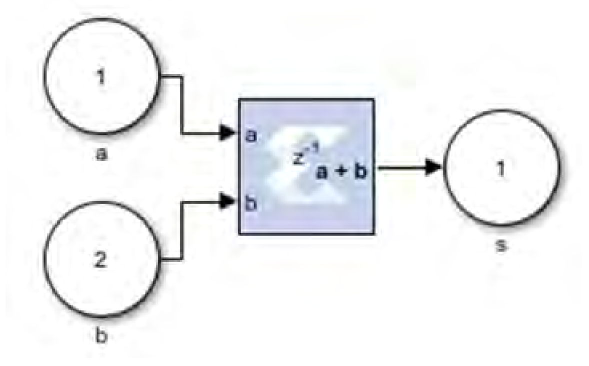
Hello World

In this example, you will run the PG API in the *learning* mode where you can type the commands in the MATLAB® command shell.

1. To start a new learning session, in MATLAB command console, run: `xBlock`.
2. Type the following three commands in MATLAB command console to create a new Subsystem named '`Subsystem`' inside a new model named '`untitled`'.

```
[a, b] = xInport('a', 'b');
s = xOutport('s');
adder = xBlock('AddSub', struct('latency', 1), {a, b}, {s});
```

Figure 474: **Subsystem Example**



The above commands create the Subsystem with two Simulink Inports `a` and `b`, an adder block having a latency of one, and a Simulink Outport `s`. The two Inports source the adder which in turn sources the Subsystem outport. The `AddSub` parameter refers to the AddSub block inside the **xbsIndex_r4** library. By default, if the full block path is not specified, `xBlock` will search `xbsIndex_r4` and built-in libraries in turn. The library must be loaded before using `xBlock`. So please use `load_system` to load the library before invoking `xBlock`.

TIP: *If you type* `adder` *in the MATLAB console, Model Composer will print a brief description of the adder block to the MATLAB console and the block is highlighted in the Simulink diagram. Similarly, you can type* `a`, `b`, *and* `s` *to highlight Subsystem Inports and Outports.*

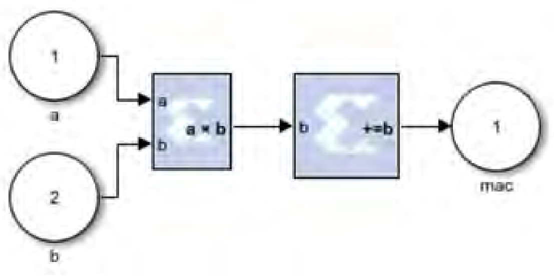
MACC

1. Run this example in the learning mode. To start a new learning session, run: `xBlock`.
2. Type the following commands in the MATLAB console window to create a multiply-accumulate function in a new Subsystem.

```
[a, b] = xInport('a', 'b');
mac = xOutport('mac');
m = xSignal;
mult = xBlock('Mult', struct('latency', 0, 'use_behavioral_HDL', 'on'),
{a, b},
{m});
acc = xBlock('Accumulator', struct('rst', 'off', 'use_behavioral_HDL',
'on'), {m},
{mac});
```

By directing Model Composer to generate behavioral HDL, the two blocks should be packed into a single DSP48 block. As of this writing, Vivado synthesis only does so if you force the multiplier block to be combinational.

Figure 475: **Forcing a Mult Block**



***Note*:** If you do not close the model that is created in example 1, example 2 is created in a model named `untiltled1`. Otherwise, a new model, `untitled`, is created for this example.

TIP: *The PG API provides functions to get information about blocks and signals in the generated Subsystem. After each of the following commands, observe the output in the MATLAB console and the effect on the Simulink diagram.*

```
mult_ins = mult.getInSignals
mult_ins{1}
mult_ins{2}
src_a = mult_ins{1}.getSrc
src_a{1}
m_dst = m.getDst
m_dst{1}
m_dst{1}.block
```

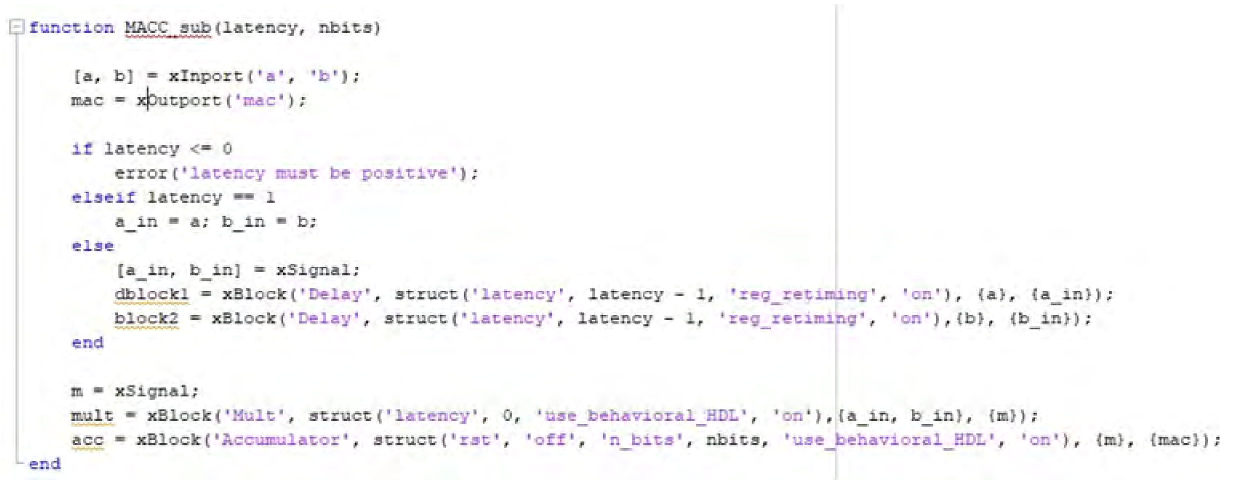
MACC in a Masked Subsystem

If you want a particular Subsystem to be generated by the PG API and pass parameters from the mask parameters of that Subsystem to PG API, you need to run the PG API in *production* mode, where you need to have a physical M-function file and pass that function to the `xBlock` constructor.

1. First create the top-level PG API M-function file `MACC_sub.m` with the following lines.

```
function MACC_sub(latency, nbits)
[a, b] = xInport('a', 'b');
mac = xOutport('mac');
if latency <= 0
   error('latency must be positive');
elseif latency == 1
   a_in = a; b_in = b;
else
   [a_in, b_in] = xSignal;
   dblock1 = xBlock('Delay', struct('latency', latency - 1,
'reg_retiming', 'on'),
{a}, {a_in});
   block2 = xBlock('Delay', struct('latency', latency - 1,
'reg_retiming', 'on'),
{b}, {b_in});
end
m = xSignal;
mult = xBlock('Mult', struct('latency', 0, 'use_behavioral_HDL', 'on'),
{a_in,
b_in}, {m});
acc = xBlock('Accumulator', struct('rst', 'off', 'n_bits', nbits,
'use_behavioral_HDL', 'on'), {m}, {mac});
```

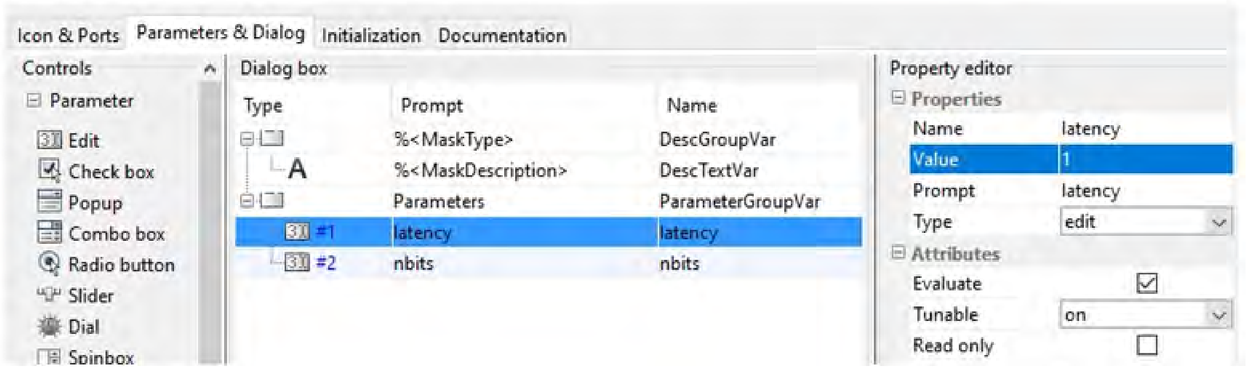
Figure 476: **Top-Level PG API M-Function File**



```
function MACC_sub(latency, nbits)

    [a, b] = xInport('a', 'b');
    mac = xOutport('mac');

    if latency <= 0
        error('latency must be positive');
    elseif latency == 1
        a_in = a; b_in = b;
    else
        [a_in, b_in] = xSignal;
        dblock1 = xBlock('Delay', struct('latency', latency - 1, 'reg_retiming', 'on'), {a}, {a_in});
        block2 = xBlock('Delay', struct('latency', latency - 1, 'reg_retiming', 'on'),{b}, {b_in});
    end

    m = xSignal;
    mult = xBlock('Mult', struct('latency', 0, 'use_behavioral_HDL', 'on'),{a_in, b_in}, {m});
    acc = xBlock('Accumulator', struct('rst', 'off', 'n_bits', nbits, 'use_behavioral_HDL', 'on'), {m}, {mac});
end
```

2. To mask the Subsystem defined by the script, add two mask parameters latency and nbits.

Figure 477: **Adding Latency Parameter**



Type	Prompt	Name
	%<MaskType>	DescGroupVar
A	%<MaskDescription>	DescTextVar
	Parameters	ParameterGroupVar
#1	latency	latency
#2	nbits	nbits

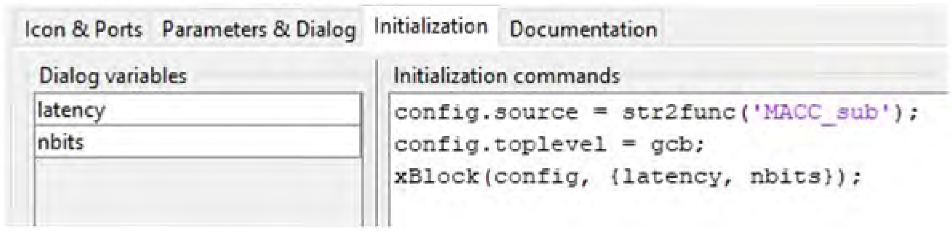
Properties	
Name	latency
Value	1
Prompt	latency
Type	edit
Attributes	
Evaluate	✓
Tunable	on
Read only	☐

3. Then put the following lines to the mask initialization of the Subsystem.

```
config.source = str2func('MACC_sub');
config.toplevel = gcb;
xBlock(config, {latency, nbits});
```

In the *production* mode, the first argument of the `xBlock` constructor is a MATLAB struct for configuration, which must have a source field and a toplevel field. The source field is a function pointer pointing to the M-function and the toplevel is a string specifying the Simulink Subsystem. If the top-level field is 1, an untitled model is created and a Subsystem inside that model is created.

Figure 478: **Adding nbits Parameter**



```
config.source = str2func('MACC_sub');
config.toplevel = gcb;
xBlock(config, {latency, nbits});
```

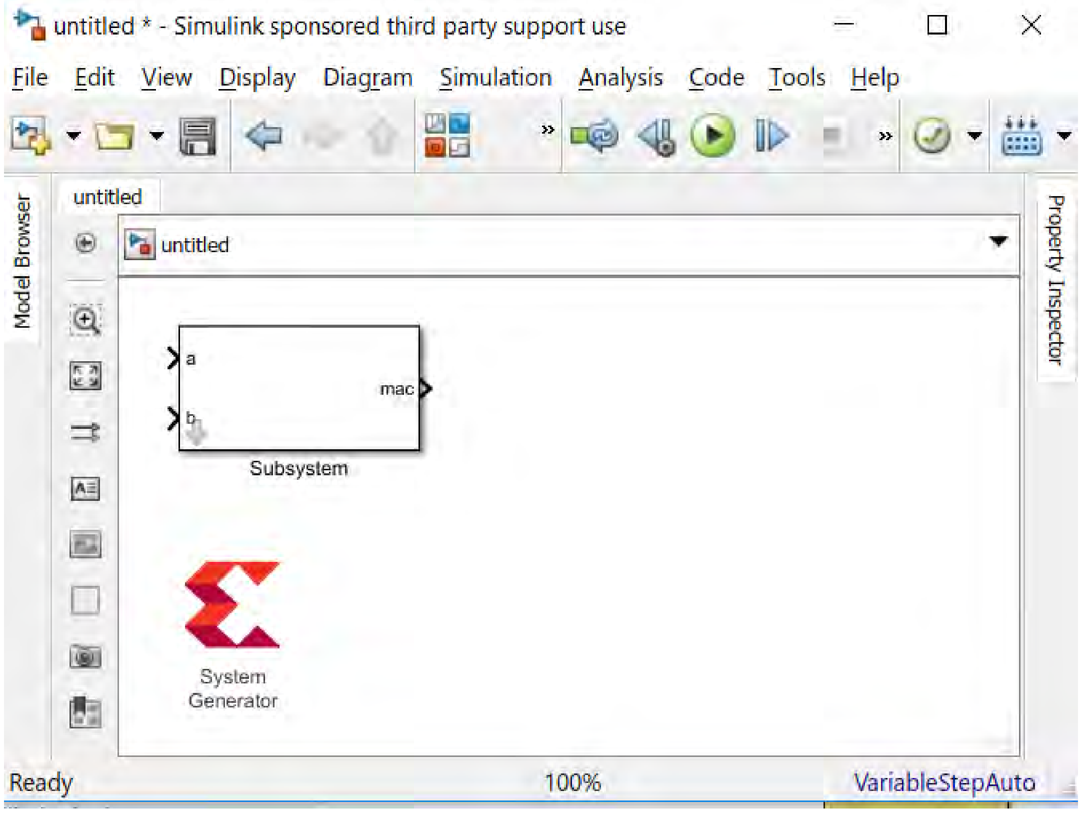
Alternatively you can use the MATLAB struct call to create the toplevel configuration:

```
xBlock(struct('source', str2func(MACC_sub), 'toplevel', gcb),{latency,
 nbits});
```

Then click **OK**.

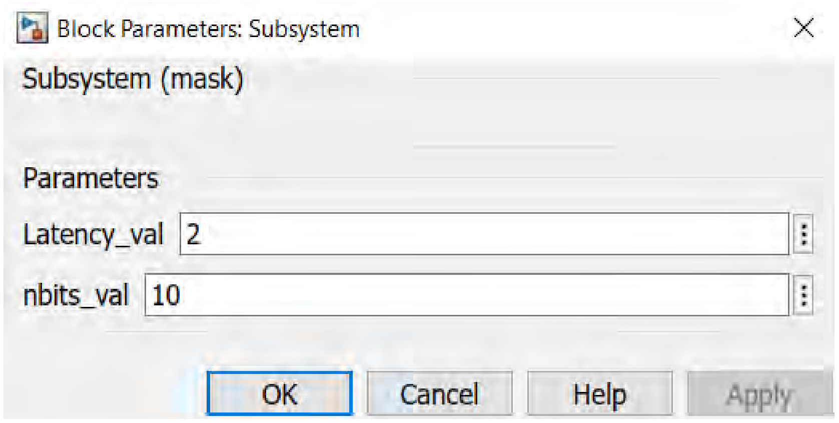
You'll get the following Subsystem.

Figure 479: **Creating Toplevel Configuration**



4. Set the mask parameters as shown in the following figure, then click **OK**:

Figure 480: **Adding Mask Parameters**



Block Parameters: Subsystem

Subsystem (mask)

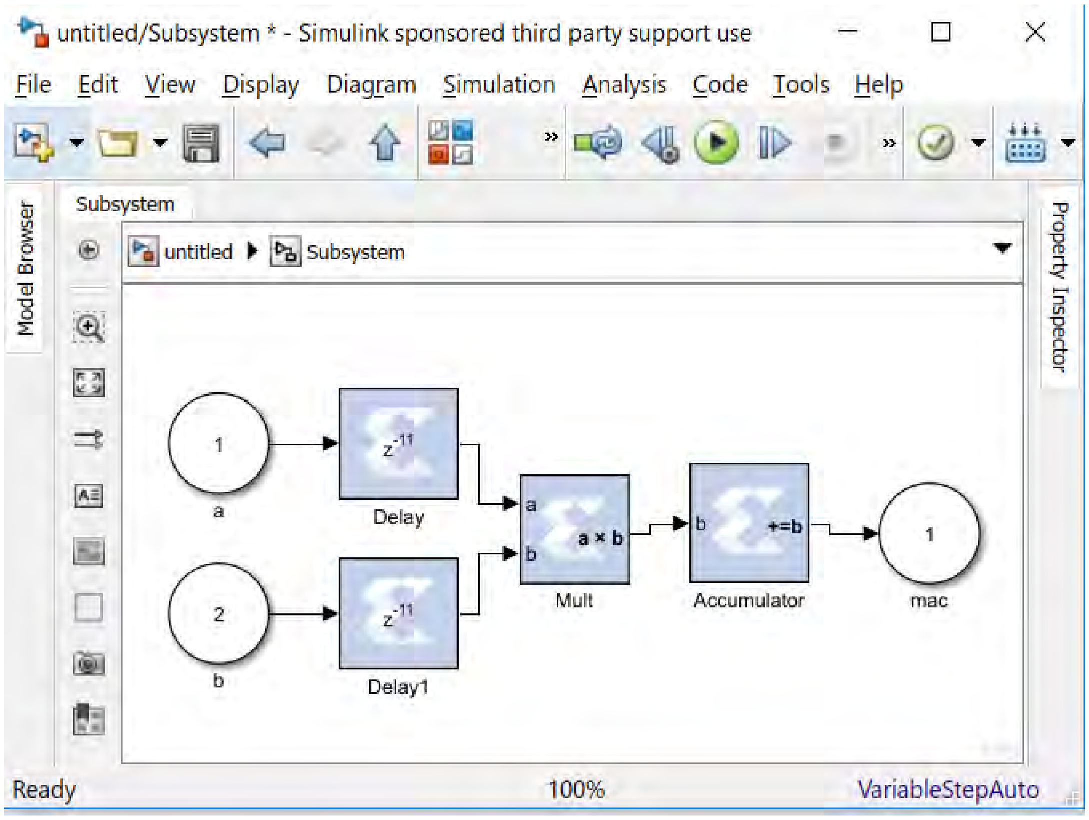
Parameters

Latency_val: 2

nbits_val: 10

The following diagram is generated:

Figure 481: **Generated Diagram**



> **TIP:** *Open `MACC_sub.m` in the MATLAB editor to debug the function. By default the `xBlock` constructor will do an auto layout in the end. If you want to see the auto layout every time a block is added, invoke the toplevel `xBlock` as the following:*

```
config.source = str2func('MACC_sub');
config.toplevel = gcb;
config.debug = 1;
xBlock(config, {latency, nbits});
```

By setting the debug field of the configuration struct to be 1, you run the PG API in debug mode where every action will trigger an auto layout.

> **TIP:** *Most often you only want to re-generate the Subsystem if needed. The `xBlock` constructor has a caching mechanism. You can specify the list of dependent files in a cell array, and set the `depend` field of the toplevel configuration with this list. If any file in the 'depend' list is changed, or the argument list that passed to the toplevel function is changed, the Subsystem is re-generated. If you want to have the caching capability for the MACC_sub, invoke the toplevel xBlock as the following:*

```
config.source = str2func('MACC_sub');
config.toplevel = gcb;
config.depend = {'MACC_sub.m'};
xBlock(config, {latency, nbits});
```

The depend field of the configuration struct is a cell array. Each element of the array is a file name. You can put a p-file name or an M-file name. You can also put a name without a suffix. The `xBlock` will use the first in the path.

PG API Error/Warning Handling and Messages

xBlock Error Messages

Condition	Error Message(s)
When calling `xBlock(NoSubSourceBlock, )` and the source block does not exist	Source block NoSubSourceBlock cannot be found.
When calling `xBlock(sourceblock, parameterBinding)`, and the parameters are illegal, xBlock will report the Illegal parameterization error. For example, `xBlock('AddSub', struct('latency', -1));`	Illegal parameterization: Latency Latency is set to a value of -1, but the value must be greater than or equal to 0
When the input port binding list contains objects other than `xSignal` or `xInport`:	Only objects of xInport or xSignal can appear in inport binding list.
When the output port binding list contains objects other than `xSignal` or `xOutport`:	Only objects of xOutport or xSignal can appear in outport binding list.
If the first argument of `xBlock` is a function pointer, the 2nd argument of `xBlock` is expected to be a cell array, otherwise, an error is thrown:	Cell array is expected for the second argument of the xBlock call
If the source configuration struct has toplevel defined, it must point to a Simulink® Subsystem and it must be a char array, otherwise, an error is thrown:	Top level must be a char array

Condition	Error Message(s)
If an object in the outport binding list has already been driven by something, for example, if you try to have two driving sources, an error is thrown.	Source of xSignal object already exists

xInport Error Messages

Condition	Error Message(s)
If you try to create an `xInport` object with the same name the second time, an error is thrown. For example, if you call p = xInport('a', 'a').	A new block named 'untitled/Subsystem/a' cannot be added.

xOutport Error Messages

Condition	Error Message(s)
If you try to create an xOutport object with the same name the second time, an error is thrown. For example, if you call p = xOutport('a', 'a').	A new block named 'untitled/Subsystem/a' cannot be added.
If you try to bind an xOutport object twice, an error is thrown. For example, the following sequence of calls will cause an error: [a, b] = xInport('a', 'b'); c = xOutport('c'); c.bind(a); c.bind(b);	The destination port already has a line connection.

xSignal Error Messages

Condition	Error Message(s)
If you try to bind an `xSignal` object with two sources, an error is thrown. For example, the following sequence of calls will cause an error: [a, b] = xInport('a', 'b'); sig = xSignal; sig.bind(a); sig.bind(b);	Source of xSignal object already exists.

xsub2script Error Messages

Condition	Error Message(s)
xlsub2script is invoked without any argument.	An argument is expected for xlsub2script
The first argument is not a Subsystem or the model is not opened.	The first argument must be a model, Subsystem, or a block. Please make sure the model is opened or the argument is a valid string for a model or a block.
A Subsystem has Simulink function calls in its mask initialization code.	Subsystem has Simulink function calls, such as gcb, get_param, set_param, add_block. Please remove these calls and run xlsub2script again or you can pick a different Subsystem to run xlsub2script.
The Subsystem has Goto blocks.	You have the following Goto blocks, please modify the model to remove them and run xlsub2script again.

M-Code Access to Hardware Co-Simulation

HDL Hardware co-simulation in Model Composer brings on-chip acceleration and verification capabilities into the Simulink simulation environment. In the typical Model Composer flow, a Model Composer model is first compiled for a hardware co-simulation platform, during which a hardware implementation (bitstream) of the design is generated and associated to a hardware co-simulation block. The block is inserted into a Simulink model and its ports are connected with appropriate source and sink blocks. The whole model is simulated while the compiled Model Composer design is executed on an FPGA.

Alternatively, it is possible to programmatically control the hardware created through the Model Composer HDL hardware co-simulation flow using MATLAB M-code (M-Hwcosim). The M-Hwcosim interfaces allow for MATLAB objects that correspond to the hardware to be created in pure M-code, independent of the Simulink framework. These objects can then be used to read and write data into hardware.

This capability is useful for providing a scripting interface to hardware co-simulation, allowing for the hardware to be used in a scripted test bench or deployed as hardware acceleration in M-code. Apart from supporting the scheduling semantics of a Model Composer simulation, M-Hwcosim also gives the flexibility for any arbitrary schedule to be used. This flexibility can be exploited to improve the performance of a simulation, if the user has apriori knowledge of how the design works. Additionally, the M-Hwcosim objects provide accessibility to the hardware from the MATLAB console, allowing for the hardware internal state to be introspected interactively.

Compiling Hardware for Use with M-Hwcosim

Compiling hardware for use in M-Hwcosim follows the same flow as the typical Model Composer HDL hardware co-simulation flow. You start off with a Model Composer model in Simulink, select a hardware co-simulation target in the System Generator token and click Generate. At the end of the generation, a hardware co-simulation library is created.

Among other files in the netlist directory, you can find a bit file and an hwc file. The bit file corresponds to the FPGA implementation, and the hwc file contains information required for M-Hwcosim. Both bit file and hwc file are paired by name, e.g. mydesign_cw.bit and mydesign_cw.hwc.

The hwc file specifies additional meta information for describing the design and the chosen hardware co-simulation interface. With the meta information, a hardware co-simulation instance can be instantiated using M-Hwcosim, through which you can interact with the co-simulation engine.

M-Hwcosim inherits the same concepts of ports and fixed point notations as found in the existing co-simulation block. Every design exposes its top-level ports for external access.

M-Hwcosim Simulation Semantics

The simulation semantics for M-Hwcosim differs from that used during hardware co-simulation in a Model Composer block diagram; the M-Hwcosim simulation semantics is more flexible and is capable of emulating the simulation semantics used in the block-based hardware co-simulation.

In the block-based hardware co-simulation, a rigid simulation semantic is imposed; before advancing a clock cycle, all the input ports of the hardware co-simulation are written to. Next all the output ports are read and the clock is advanced. In M-Hwcosim the scheduling of when ports are read or written to, is left to the user. For instance it would be possible to create a program that would only write data to certain ports on every other cycle, or to only read the outputs after a certain number of clock cycles. This flexibility allows users to optimize the transfer of data for better performance.

Data Representation

M-Hwcosim uses fixed point data types internally, while it consumes and produces double precision floating point values to external entities. All data samples passing through a port are fixed point numbers. Each sample has a preset data width and an implicit binary point position that are fixed at the compilation time. Data conversions (from double precision to fixed point) happen on the boundary of M-Hwcosim. In the current implementation, quantization of the input data is handled by rounding, and overflow is handled by saturation.

Interfacing to Hardware from M-Code

When a model has been compiled for hardware co-simulation, the generated bitstream can be used in both a model-based Simulink flow, or in M-code executed in MATLAB. The general sequence of operations to access a bitstream in hardware typically follows the sequence described below.

1. Configure the hardware co-simulation interface. Note that the hardware co-simulation configuration is persistent and is saved in the `hwc` file. If the co-simulation interface is not changed, there is no need to re-run this step.
2. Create a M-Hwcosim instance for a particular design.
3. Open the M-Hwcosim interface.
4. Repeatedly run the following sub-steps until the simulation ends.
5. Write simulation data to input ports.
6. Read simulation data from output ports.
7. Advance the design clock by one cycle.
8. Close the M-Hwcosim interface.
9. Release the M-Hwcosim instance.

Automatic Generation of M-Hwcosim Testbench

M-Hwcosim enables the test bench generation for hardware co-simulation. When the **Create testbench** option is checked in the System Generator token, the hardware co-simulation compilation flow generates an M-code script (`<design>_hwcosim_test.m`) and golden test data files (`<design>_<port>_hwcosim_test.dat`) for each gateway based on the Simulink simulation. The M-code script uses the M-Hwcosim API to implement a test bench that simulates the design in hardware and verifies the results against the golden test data. Any simulation mismatch is reported in a result file (`<design>_hwcosim_test.results`).

As shown below in the Example, the test bench code generated is easily readable and can be used as a basis for your own simulation code.

```
                function multi_rates_cw_hwcosim_test
                try
                % Define the number of hardware cycles for the simulation.
                ncycles = 10;

                % Load input and output test reference data.
                testdata_in2 = load('multi_rates_cw_in2_hwcosim_test.dat');
                testdata_in3 = load('multi_rates_cw_in3_hwcosim_test.dat');
                testdata_in7 = load('multi_rates_cw_in7_hwcosim_test.dat');
                testdata_pb00 =
load('multi_rates_cw_pb00_hwcosim_test.dat');
                testdata_pb01 =
load('multi_rates_cw_pb01_hwcosim_test.dat');
                testdata_pb02 =
load('multi_rates_cw_pb02_hwcosim_test.dat');
                testdata_pb03 =
load('multi_rates_cw_pb03_hwcosim_test.dat');
                testdata_pb04 =
load('multi_rates_cw_pb04_hwcosim_test.dat');

                % Pre-allocate memory for test results.
                result_pb00 = zeros(size(testdata_pb00));
                result_pb01 = zeros(size(testdata_pb01));
                result_pb02 = zeros(size(testdata_pb02));
                result_pb03 = zeros(size(testdata_pb03));
                result_pb04 = zeros(size(testdata_pb04));

                % Initialize sample index counter for each sample period to
be
                % scheduled.
                insp_2 = 1;
                insp_3 = 1;
                insp_7 = 1;
                outsp_1 = 1;
                outsp_2 = 1;
                outsp_3 = 1;
                outsp_7 = 1;

                % Define hardware co-simulation project file.
                project = 'multi_rates_cw.hwc';

                % Create a hardware co-simulation instance.
                h = Hwcosim(project);
```

```
                     % Open the co-simulation interface and configure the
hardware.
                     try
                     open(h);
                     catch
                     % If an error occurs, launch the configuration GUI for the
user
                     % to change interface settings, and then retry the process
again.
                     release(h);
                     drawnow;
                     h = Hwcosim(project);
                     open(h);
                     end

                     % Simulate for the specified number of cycles.
                     for i = 0:(ncycles-1)

                     % Write data to input ports based their sample period.
                     if mod(i, 2) == 0
                     h('in2') = testdata_in2(insp_2);
                     insp_2 = insp_2 + 1;
                     end
                     if mod(i, 3) == 0
                     h('in3') = testdata_in3(insp_3);
                     insp_3 = insp_3 + 1;
                     end
                     if mod(i, 7) == 0
                     h('in7') = testdata_in7(insp_7);
                     insp_7 = insp_7 + 1;
                     end

                     % Read data from output ports based their sample period.
                     result_pb00(outsp_1) = h('pb00');
                     result_pb04(outsp_1) = h('pb04');
                     outsp_1 = outsp_1 + 1;
                     if mod(i, 2) == 0
                     result_pb01(outsp_2) = h('pb01');
                     outsp_2 = outsp_2 + 1;
                     end
                     if mod(i, 3) == 0
                     result_pb02(outsp_3) = h('pb02');
                     outsp_3 = outsp_3 + 1;
                     end
                     if mod(i, 7) == 0
                     result_pb03(outsp_7) = h('pb03');
                     outsp_7 = outsp_7 + 1;
                     end

                     % Advance the hardware clock for one cycle.
                     run(h);

                     end

                     % Release the hardware co-simulation instance.
                     release(h);

                     % Check simulation result for each output port.
                     logfile = 'multi_rates_cw_hwcosim_test.results';
                     logfd = fopen(logfile, 'w');
                     sim_ok = true;
                     sim_ok = sim_ok & check_result(logfd, 'pb00',
     testdata_pb00, result_pb00);
                     sim_ok = sim_ok & check_result(logfd, 'pb01',
     testdata_pb01, result_pb01);
                     sim_ok = sim_ok & check_result(logfd, 'pb02',
     testdata_pb02, result_pb02);
                     sim_ok = sim_ok & check_result(logfd, 'pb03',
     testdata_pb03, result_pb03);
                     sim_ok = sim_ok & check_result(logfd, 'pb04',
     testdata_pb04, result_pb04);
                     fclose(logfd);
                     if ~sim_ok
                     error('Found errors in simulation results. Please refer to
     ''%s'' for details.',
                     logfile);
                     end

                     catch
                     err = lasterr;
                     try release(h); end
                     error('Error running hardware co-simulation testbench. %s',
     err);
                     end


     %----------------------------------------------------------------------

                     function ok = check_result(fd, portname, expected, actual)
                     ok = false;

                     fprintf(fd, ['\n' repmat('=', 1, 95), '\n']);
                     fprintf(fd, 'Output: %s\n\n', portname);

                     % Check the number of data values.
                     nvals_expected = numel(expected);
                     nvals_actual = numel(actual);
                     if nvals_expected ~= nvals_actual
                     fprintf(fd, ['The number of simulation output values (%d)
     differs ' ...
                     'from the number of reference values (%d).\n'], ...
                     nvals_actual, nvals_expected);
                     return;
                     end

                     % Check for simulation mismatches.
                     mismatches = find(expected ~= actual);
                     num_mismatches = numel(mismatches);
                     if num_mismatches > 0
                     fprintf(fd, 'Number of simulation mismatches = %d\n',
     num_mismatches);
                     fprintf(fd, '\n');
                     fprintf(fd, 'Simulation mismatches:\n');
                     fprintf(fd, '----------------------\n');
                     fprintf(fd, '%10s %40s %40s\n', 'Cycle', 'Expected values',
     'Actual values');
                     fprintf(fd, '%10d %40.16f %40.16f\n', ...
                     [mismatches-1; expected(mismatches); actual(mismatches)]);
                     return;
                     end

                     ok = true;
                     fprintf(fd, 'Simulation OK\n');
```

Resource Management

M-Hwcosim manages resources that it holds for a hardware co-simulation instance. It releases the held resources upon the invocation of the release instruction or when MATLAB exits. However, it is recommended to perform an explicit cleanup of resources when the simulation finishes or throws an error. To allow proper cleanup in case of errors, it is suggested to enclose M-Hwcosim instructions in a MATLAB try-catch block as illustrated below.

```
try
% M-Hwcosim instructions here
catch
err = lasterror;
% Release any Hwcosim instances
try release(hwcosim_instance); end
rethrow(err);
end
```

The following command can be used to release all hardware co-simulation instances.

```
xlHwcosim('release');       % Release all Hwcosim instances
```

M-Hwcosim MATLAB Class

The `Hwcosim` MATLAB class provides a higher level abstraction of the hardware co-simulation engine. Each instantiated Hwcosim object represents a hardware co-simulation instance. It encapsulates the properties, such as the unique identifier, associated with the instance. Most of the instruction invocations take the Hwcosim object as an input argument. For further convenience, alternative shorthand is provided for certain operations.

Actions	Syntax
Constructor	`h = Hwcosim(project)`
Destructor	`release(h)`
Open hardware	`open(h)`
Close hardware	`close(h)`
Write data	`write(h, 'portName', inData);`
Read data	`outData = read(h, 'portName');`
Run	`run(h);`
Port information	`portinfo(h);`
Set property	`set(h, 'propertyName', propertyValue);`
Get property	`propertyValue = get(h, 'propertyName');`

Constructor

Syntax

```
h = Hwcosim(project);
```

Description

Creates an Hwcosim instance. Note that an instance is a reference to the hardware co-simulation project and does not signify an explicit link to hardware; creating a Hwcosim object informs the Hwcosim engine where to locate the FPGA bitstream, it does not download the bitstream into the FPGA. The bitstream is only downloaded to the hardware after an open command is issued.

The project argument should point to the hwc file that describes the hardware co-simulation.

Creating the Hwcosim object will list all input and output ports. The example below shows the output of a call to the Hwcosim constructor, displaying the ID of the object and a list of all the input and output gateways/ports.

```
>> h = Hwcosim(p)
Model Composer HDL Hardware Co-simulation Object
  id: 30247
  inports:
      gateway_in
      gateway_in2
  outports:
      gateway_out
```

Destructor

Syntax

```
release(h);
```

Description

Releases the resources used by the Hwcosim object h. If a link to hardware is still open, release will first close the hardware.

Open Hardware

Syntax

```
open(h);
```

Description

Opens the connection between the host PC and the FPGA. Before this function can be called, the hardware co-simulation interface must be configured. The argument, h, is a handle to an Hwcosim object.

Close hardware

Syntax

```
close(h);
```

Description

Closes the connection between the host PC and the FPGA. The argument, h, is a handle to an Hwcosim object.

Write data

Syntax

```
h('portName') = inData; %If inData is array, results in burst write.

h('portName') = [1 2 3 4];

write(h, 'portName', inData);

write(h, 'portName', [1 2 3 4]); %burst mode
```

Description

Ports are referenced by their legalized names. Name legalization is a requirement for VHDL and Verilog synthesis, and converts names into all lower-case, replaces white space with underscores, and adds unique suffixes to avoid namespace collisions. To find out what the legalized input and output port names are, run the helper command `portinfo(h)`, or see the output of Hwcosim at the time of instance creation.

`inData` is the data to be written to the port. Normal single writes are performed if `inData` is a scalar value. If burst mode is enabled and `inData` is a 1xn array, it will be interpreted as a timeseries and written to the port via burst data transfer.

Read data

Syntax

```
outData = h('portName');

outData = h('portName', 25); %burst mode
outData = read(h, 'portName');

outData = read(h, 'portName', 25); %burst
```

Description

Ports are referenced by their legalized names (see previous class above).

If burst mode is enabled, and depending on whether the read command has 3 or 4 parameters (2 or 3 parameters in the case of a subscript reference `h('portName', ...)`), `outData` will be assigned a scalar or a 1xn array. If an array, the data is the result of a burst data transfer.

Run

Syntax

```
run(h);

run(h, n);

run(h, inf); %start free-running clock

run(h, 0); %stop free-running clock
```

Description

When the hardware co-simulation object is configured to run in single-step mode, the run command is used to advance the clock. run(h) will advance the clock by one cycle. run(h,n) will advance the clock by n cycles.

The run command is also used to turn on (and off) free-running clock mode: `run(h, inf)` will start the free-running clock and `run(h, 0)` will stop it.

A read of an output port will need to be preceded either by a 'dummy' run command or by a write, in order to force a synchronization of the read cache with the hardware.

Port Information

Syntax

```
portinfo(h);
```

Description

This method will return a MATLAB struct array with fields `inports` and `outports`, which themselves are struct arrays holding all input and output ports, respectively, again represented as struct arrays. The fieldnames of the individual port structs are the legalized portnames themselves, so you may obtain a cell array of input port names suitable for Hwcosim `write` commands by issuing these commands:

```
a = portinfo(h);
inports = fieldnames(a.inports);
```

You can issue a similar series of commands for output ports (`outports`).

Additional information contained in the port structs are `simulink_name`, which provides the fully hierarchical Simulink name including spaces and line breaks, `rate`, which contains the signal's rate period with respect to the DUT clock, `type`, which holds the Model Composer data type information, and, if burst mode is enabled, `fifo_depth`, indicating the maximum size of data bursts that can be sent to Hardware in a batch.

Set property

Syntax

```
set(h, 'propertyName', propertyValue);
```

Description

The set method sets or changes any of the contents of the internal properties table of the Hwcosim instance `h`. It is required that `h` already exists before calling this method.

Examples

```
set(h, 'booleanProperty', logical(0));

set(h, 'integerProperty', int32(12345));

set(h, 'doubleProperty', pi);

set(h, 'stringProperty', 'Rosebud!');
```

Get property

Syntax

The get property method returns the value of any of the contents of the internal properties table in the Hwcosim instance `h`, referenced by the propertyName key. It is required that `h` already exists before calling this method. If the `propertyName` key does not exist in h, the method throws an exception and prints an error message.

Examples

```
bool_val = get(h, 'booleanProperty');

int_val = get(h, 'integerProperty');

double = get(h, 'doubleProperty');

str_val = get(h, 'stringProperty');
```

M-Hwcosim Utility Functions

xlHwcosim

Syntax

```
xlHwcosim('release');
```

Description

When M-Hwcosim objects are created global system resources are used to register each of these objects. These objects are typically freed when a release command is called on the object. xlHwcosim provides an easy way to release all resources used by M-Hwcosim in the event of an unexpected error. The release functions for each of the objects should be used if possible because the xlHwcosim call release the resources for all instances of a particular type of object.

Example

```
xlHwcosim('release') %release all instances of Hwcosim objects.
```

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note**:** For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

1. *Introduction to FPGA Design with Vivado High-Level Synthesis* (UG998)
2. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
3. *UltraFast Vivado HLS Methodology Guide* (UG1197)
4. *Vivado Design Suite User Guide: Designing with IP* (UG896)
5. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)
6. *Versal ACAP AI Engine Programming Environment User Guide* (UG1076)
7. *Vitis Model Composer Tutorial* (UG1498)
8. *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893)
9. *Vivado Design Suite User Guide: Design Flows Overview* (UG892)
10. *ISE to Vivado Design Suite Migration Guide* (UG911)
11. *Vivado Design Suite User Guide: Using Constraints* (UG903)
12. *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894)
13. *Vivado Design Suite Tutorial: Design Flows Overview* (UG888)
14. *Vivado Design Suite User Guide: System-Level Design Entry* (UG895)
15. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973)
16. *UltraFast Design Methodology Guide for Xilinx FPGAs and SoCs* (UG949)
17. Vivado® Design Suite Documentation
18. Mathworks® Simulink® Documentation

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright