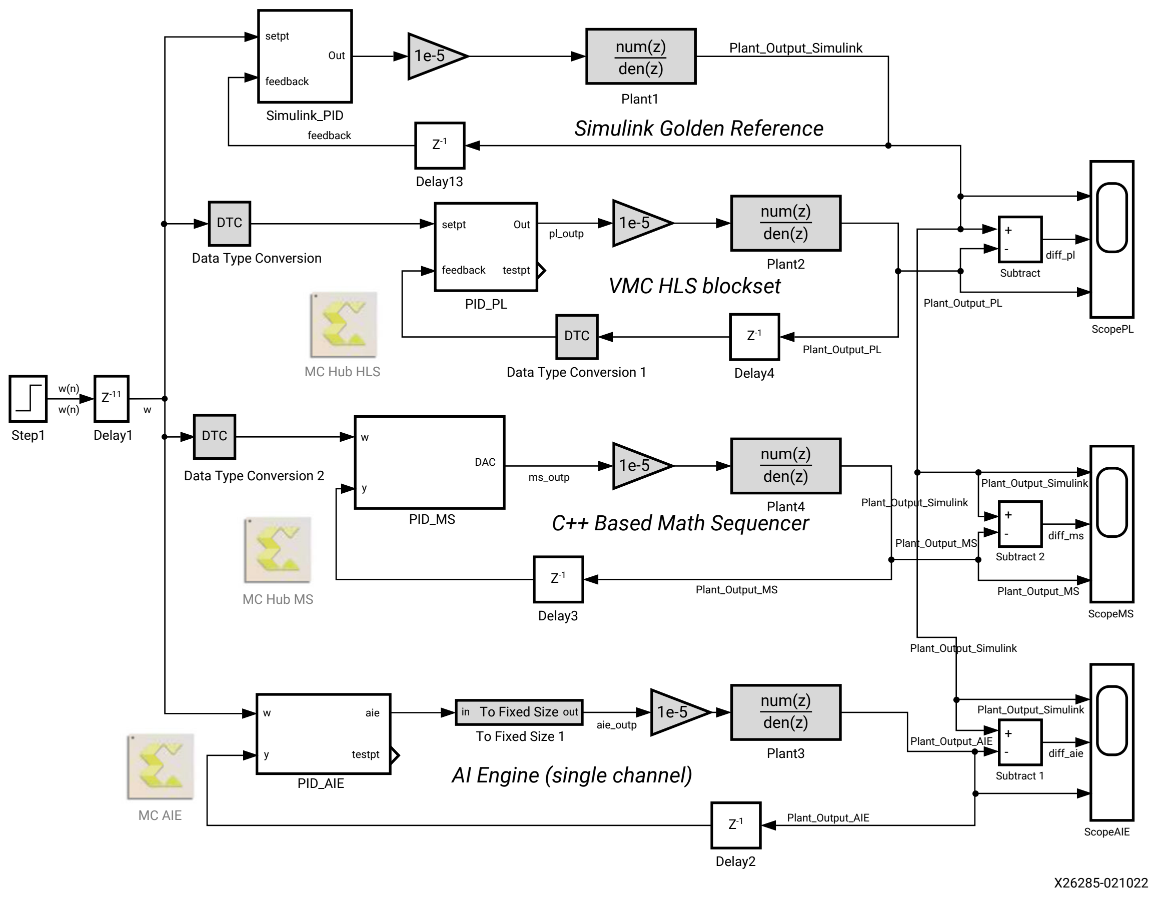
PID Controller Design with Model Composer for Versal ACAPs

XAPP1376 (v1.0) March 9, 2022

Summary

This application note is an extension to the *PID Controller Design with Model Composer* (XAPP1341), which targets the Versal® Adaptive Compute and Acceleration Platform (ACAP). With the introduction of the Versal AI Core series, Xilinx® customers have the option to perform native single precision floating-point (SPFP) operations in either the programmable logic (PL) or the AI Engines. This update demonstrates the Vitis™ Model Composer's (VMC) flexibility to implement a floating point digital signal processing (DSP) algorithm targeting either a PL or an AI Engine implementation. A known golden reference Simulink® PID model demonstrates an independent method to validate and debug results for the two different PL implementations: VMC HLS blockset or C++ Based Math Sequencer and a single channel AI Engine implementation as shown in the following figure.

Figure 1: **Multiple Approaches to Modeling an SPFP, Closed-Loop Control System**



Download the reference design files for this application note from the Xilinx website. For detailed information about the design files, see Reference Design.

Introduction

The Versal AI Core DSP58s provide SPFP operations more efficiently than the earlier 16 nm devices, as shown in the following figure and table (see *Versal AI Core Series Data Sheet: DC and AC Switching Characteristics* (DS957)).

Figure 2: **DSP58 Single Precision Floating Point Support**

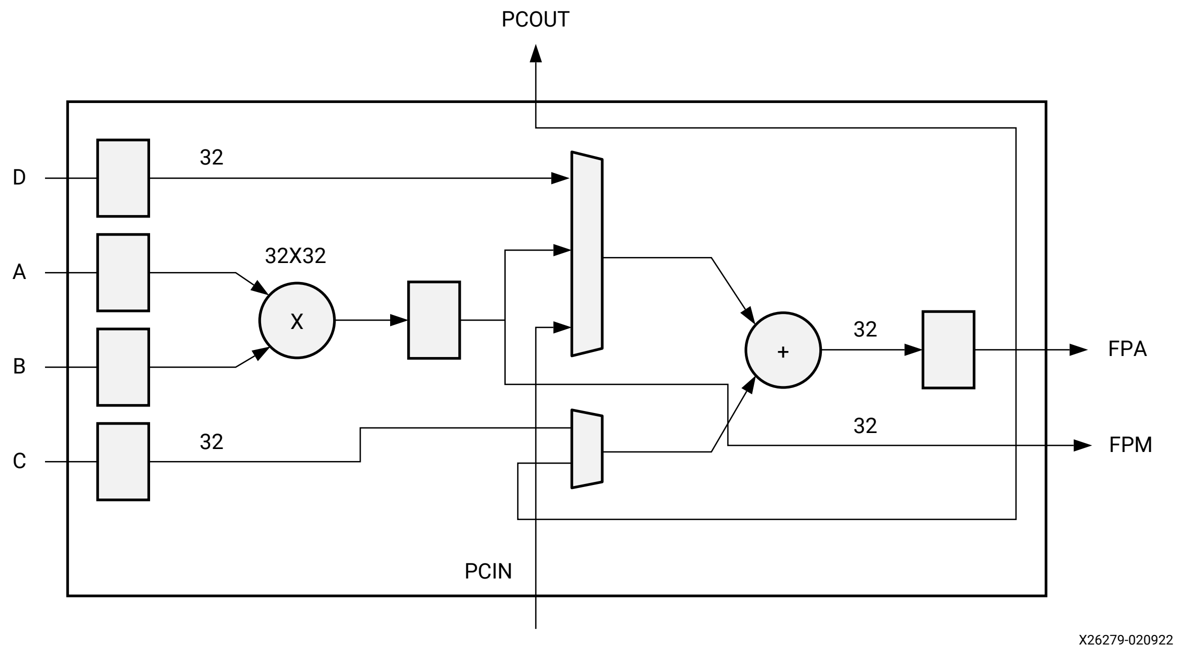


Table 1: **Direct Instantiation of Floating Point Functions**

Symbol	Description	Performance as a Function of Speed Grade and Operating Voltage (V_{CCINT})						Units
		0.88V (H)		0.80V (M)		0.70V (L)		
		-3	-2	-2	-1	-2	-1	
Floating Point Arithmetic								
F_{MAX_FP}	Floating-point operations	805	805	750	700	532	476	MHz

Prior DSP48-based devices used the integer unit and a combination of DSP48 and PL to build single, double, or custom precision floating point operators (FPO). The new DSP58 is backwards compatible with the DSP48 but adds a hard macro, an SPFP (aka FP32) multiplier, and an adder. The FP32 multiplier and adder have the following features:

- Support for cascading
- A, B, C, and D inputs 32-bit SPFP
- Adder and multiplier output 32 bits
- Both outputs available simultaneously

The SPFP multiplier and adder are IEEE-754 and OpenCL™ compliant and have the following inherent characteristics:

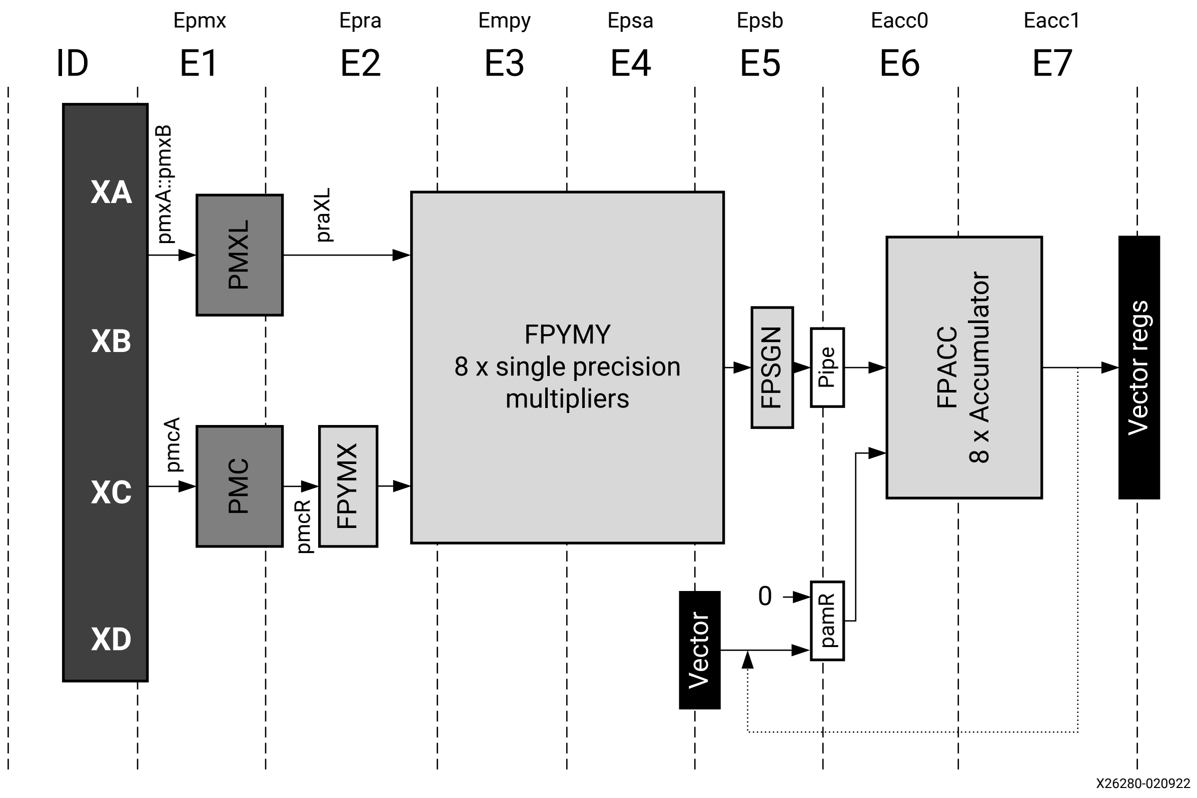
- Multiply-Add, Multiply-Subtract, Multiply-Acc
- Multiplication
- Addition

- Subtraction
- Round-towards-nearest-even
- Overflow, underflow, and invalid flags
- Instantiation and FPO IP core
- Four times more efficient SPFP implementation vs 16 nm DSP48 with no additional PL support logic required

Further, the addition of AI Engines gives the software a programmable, deterministic, and dedicated SPFP processing data path as demonstrated in the following figure. The vector processor has a dedicated floating point data path the following capabilities:

- Single precision
- Eight multiply-accumulate per cycle
- Sign change (FPSGN) is on per-lane basis

Figure 3: **AI Engine Single Precision Floating Point Support Unit**



Whether combined or independent, the DSP58 and the AI Engine high performance compute engines enhance any DSP-centric design. But implementing, debugging, and validating a Versal ACAP design can be challenging with multiple design entry approaches such as RTL, C/C++ for PL, AI Engine, and intrinsics or AI Engine APIs for AI Engine. The VMC simplifies the AI Core DSP development by assisting in the following tasks:

- DSP test bench development through pre-built Simulink toolboxes or MATLAB® source code

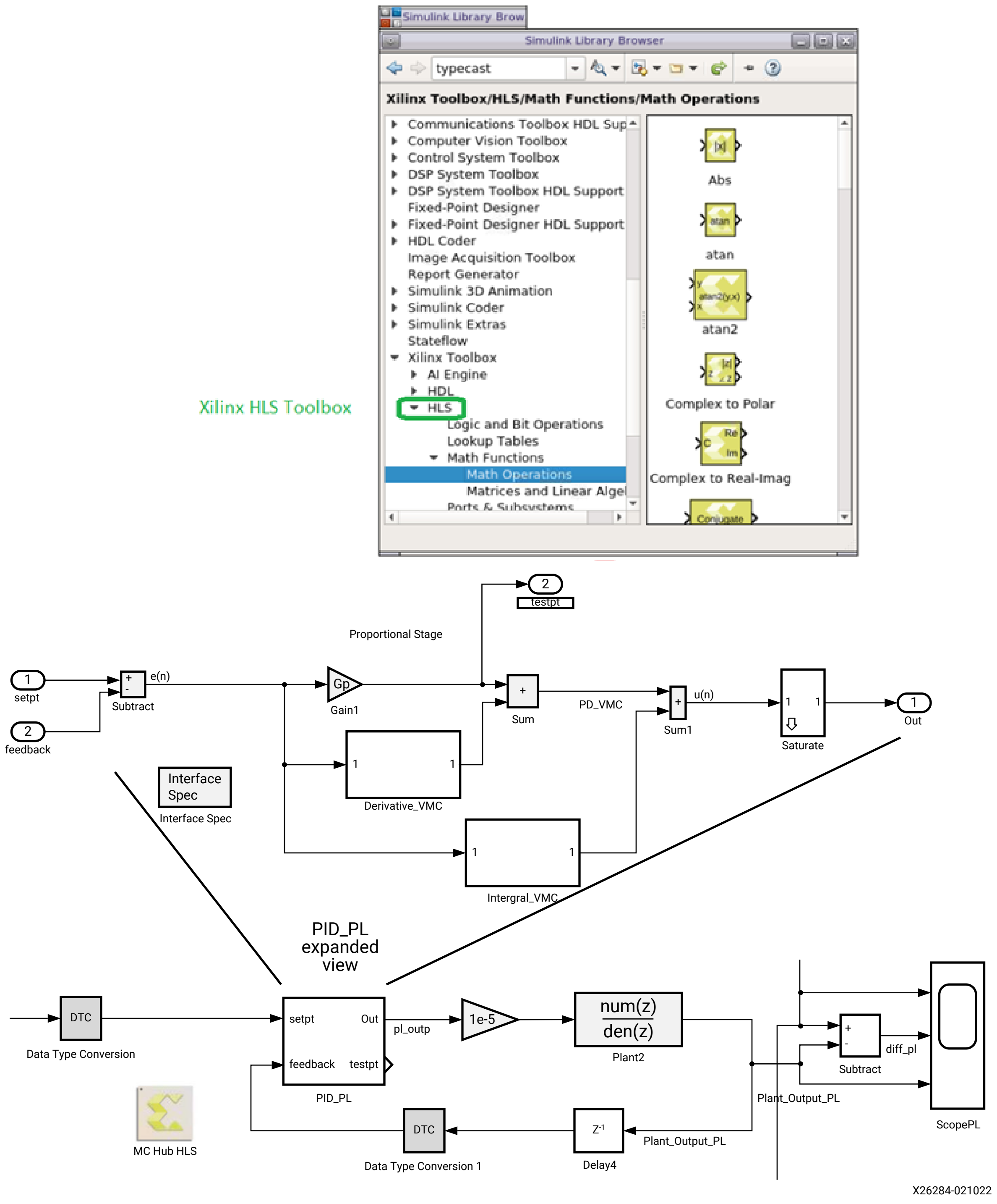
- DSP verification and validation by taking advantage of the many Simulink visualization and debug methodologies
- Node by node in situ comparison of a golden reference model to the algorithm in development
- Co-simulation and development of the mixed language model designs using C++ for PL or AI Engines, LogiCORE™ IP, RTL, and intrinsics or AI Engine APIs for AI Engines
- Functional debugging for reduced development cycles and cycle approximate AI Engine simulations
- Evaluation and exportation of a C++ design and test bench as a Vitis HLS project for resource and timing optimization
- Creation of an automated Adaptive Dataflow Graph for AI Engine designs
- AI Engine hardware validation targeting a VCK190

To demonstrate DSP development using VMC, this application note provides three examples of methodologies to implement the same SPFP PID algorithm. All models discussed in this application note use fast bit accurate C++ models for functional verification and debug for the Versal AI Core designs. It also provides a node for node comparison to a Simulink golden reference model.

HLS Toolbox Based PID Functional Simulation

The following figure is an SPFP PID implemented in PL using the VMC HLS toolbox.

Figure 4: **Single Precision Floating Point PID Using Native VMC Blocksets**

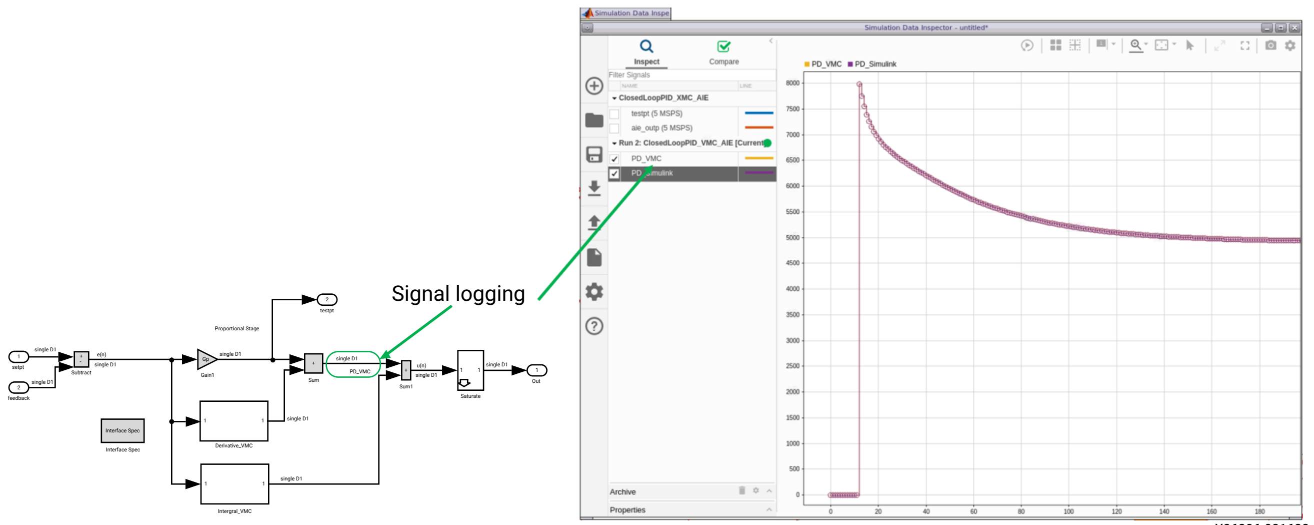


This allows the ability to compare and contrast the Simulink golden reference model to the VMC SPFP HLS toolbox implementation using Simulink tools such as scopes, display, signal logging, and more. The following figure demonstrates how to log, visualize, and compare signals in the design using:

- Simulation data inspector
- Scope

- Display
- To workspace
- Signal logging
- Port value displays

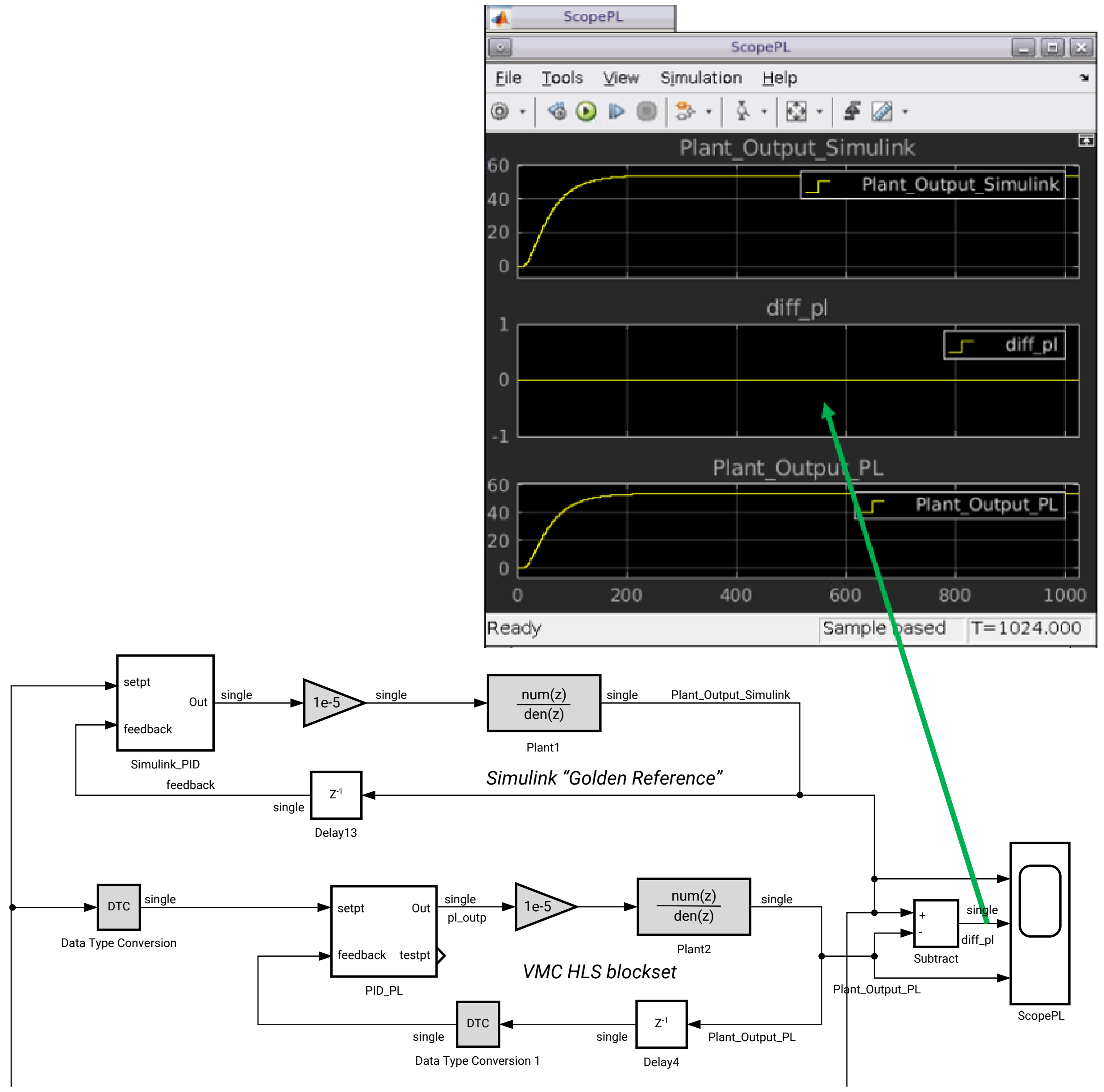
Simulink and PL Simulation Comparison



X26286-021622

In the following figure, a simple Simulink subtract is used to show zero differences between the golden reference and HLS toolbox implementation.

Figure 5: **Simulink and PL Simulation Comparison**

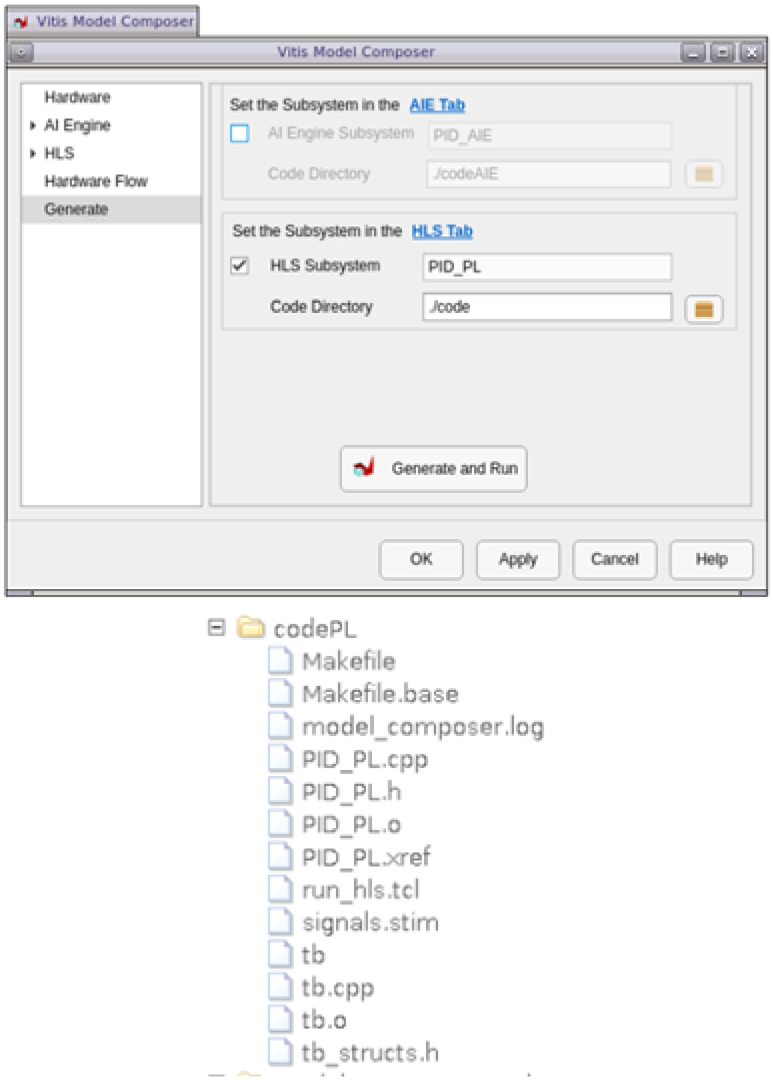


X26287-021022

HLS Toolbox Based PID Implementation

The create and run test bench option in the HLS Toolbox PL implementation VMC auto generates all files necessary to create a Vitis™ HLS project. The following figure is a screen capture from a Vitis HLS PID_project that includes a pass/fail test bench using the Simulink source stimulus.

Figure 6: **Generate a Vitis HLS HLS PID_PL Project**



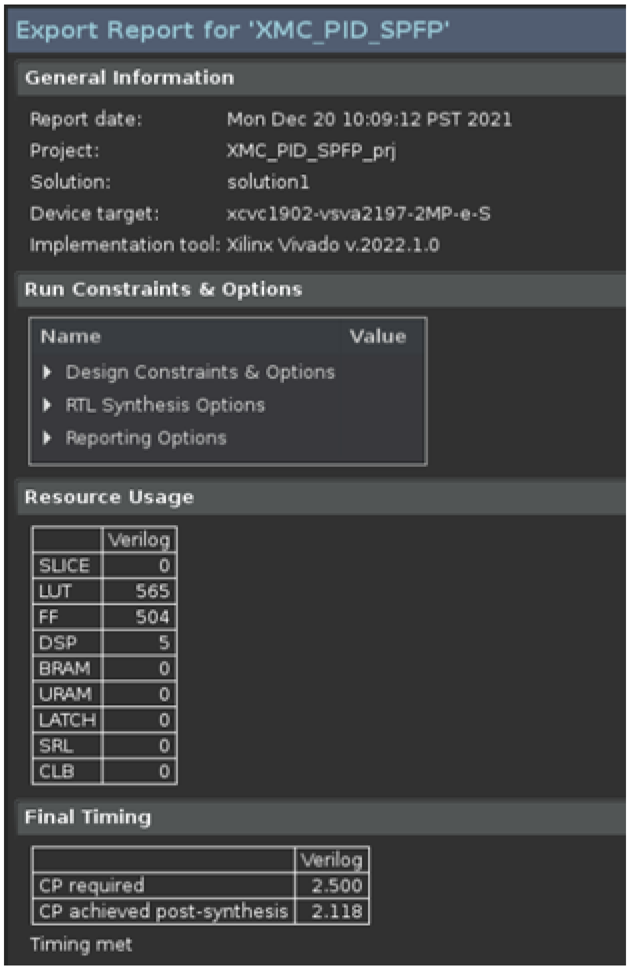
```
codePL
    Makefile
    Makefile.base
    model_composer.log
    PID_PL.cpp
    PID_PL.h
    PID_PL.o
    PID_PL.xref
    run_hls.tcl
    signals.stim
    tb
    tb.cpp
    tb.o
    tb_structs.h
```

VMC auto generates the following Vitis HLS files:

- A C++ based PID model
- A test bench
- A Vitis HLS `run_hls.tcl` file that creates a project and runs through csim and cosim

Executing the Tcl script from a Vitis HLS command prompt creates a Vitis HLS project. For GUI-based exploration and optimization, the Vitis GUI can be opened. Otherwise, continue with the Tcl script. The resource, achievable clock rate, and latency information for the HLS Toolbox implementation is demonstrated in the following figures where Clock Rate = 1/2.118e−8 = 472 MHz and Sample Rate = 1/(69 × 2.118e−9) = 6.8 MSPS.

Figure 7: **Export Report**



Export Report for 'XMC_PID_SPFP'

General Information

Report date:	Mon Dec 20 10:09:12 PST 2021
Project:	XMC_PID_SPFP_prj
Solution:	solution1
Device target:	xcvc1902-vsva2197-2MP-e-S
Implementation tool:	Xilinx Vivado v.2022.1.0

Run Constraints & Options

Name	Value
▸ Design Constraints & Options	
▸ RTL Synthesis Options	
▸ Reporting Options	

Resource Usage

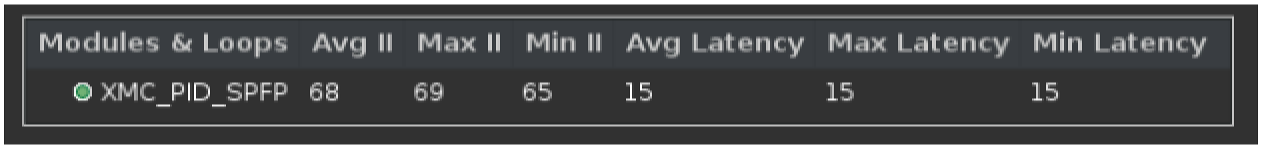
	Verilog
SLICE	0
LUT	565
FF	504
DSP	5
BRAM	0
URAM	0
LATCH	0
SRL	0
CLB	0

Final Timing

	Verilog
CP required	2.500
CP achieved post-synthesis	2.118

Timing met

Figure 8: **HLS Toolbox PID Single Precision Floating Point HLS Implementation Results**

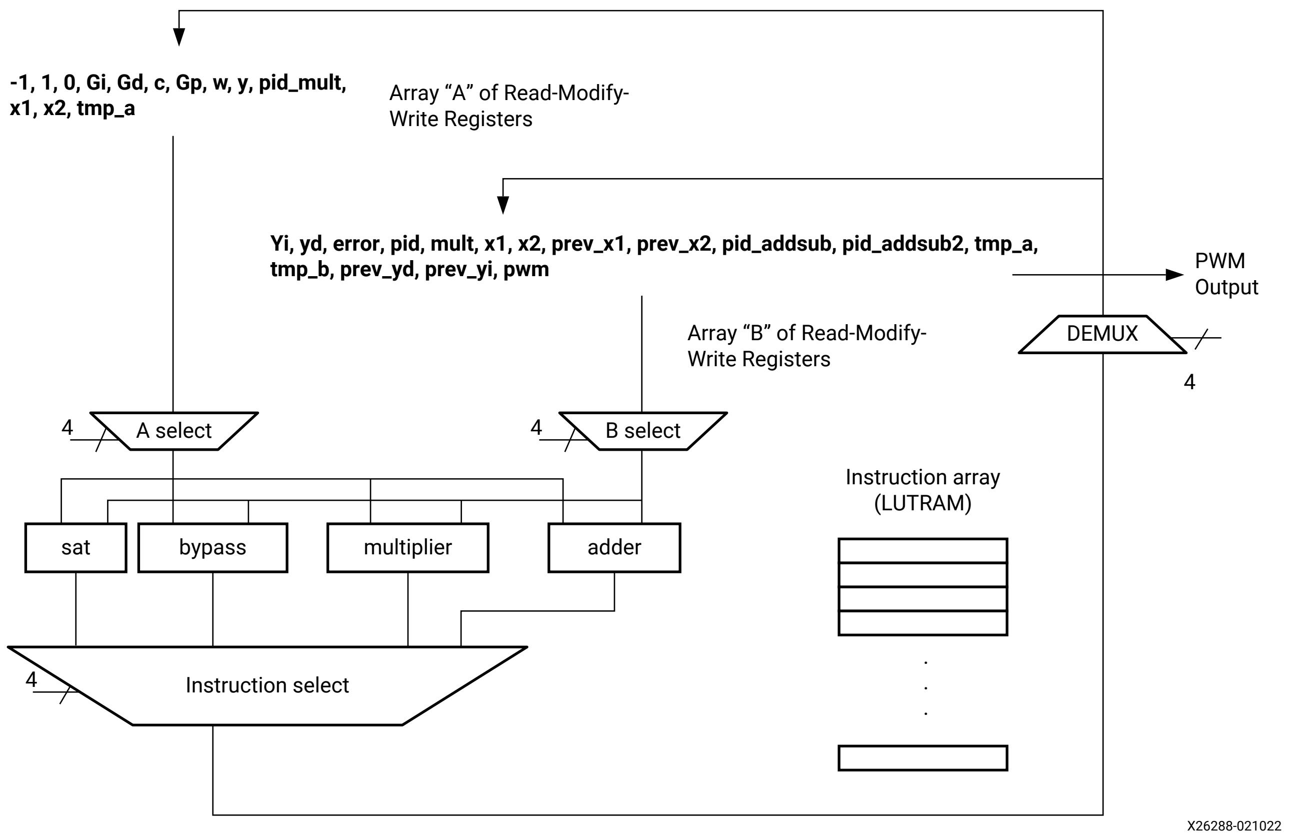


Modules & Loops	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
● XMC_PID_SPFP	68	69	65	15	15	15

VMC PL Math Sequencer Design

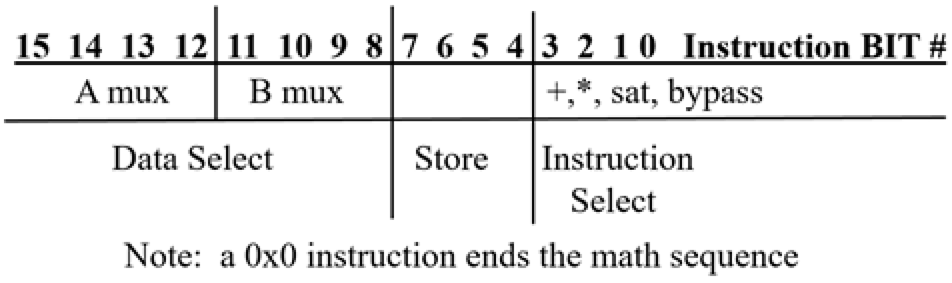
It is possible to use C or C++ to describe a control algorithm while concurrently changing the algorithm on the fly. For example, PID, PI, or lead-lag type controllers can be described using C and C++. Control algorithms are simply a series of arithmetic operations such as multiplication, addition, subtraction, saturation, and division. C and C++ easily describe a state machine that has memory (for intermediate data and instruction storage), inputs (like the W and Y inputs needed for the PID controller example), math operators, and output (PWM duty cycle) that are used to drive a DAC to control a servo motor. The arrival of input data, like W and Y inputs for the PID control loop, can be used to kick off the math sequence functionality while the end of the math sequence could be used to update a PWM. The following figure provides an example of a math sequencer optimized for a PID algorithm.

Figure 9: **Math Sequencer Block Diagram**


-1, 1, 0, Gi, Gd, c, Gp, w, y, pid_mult, x1, x2, tmp_a
Array “A” of Read-Modify-Write Registers
Yi, yd, error, pid, mult, x1, x2, prev_x1, prev_x2, pid_addsub, pid_addsub2, tmp_a, tmp_b, prev_yd, prev_yi, pwm
PWM Output
DEMUX
4
Array “B” of Read-Modify-Write Registers
4
A select
4
B select
Instruction array (LUTRAM)
sat
bypass
multiplier
adder
4
Instruction select
X26288-021022

An instruction array stored in LUTRAM controls the A and B multiplexer selects, where the resulting arithmetic operation needs to be stored in read-modify-write registers. The instruction array also controls what specific SPFP arithmetic operation (saturation, bypass, multiply, or add) is required. An instruction bit breakdown for the proposed math sequencer is shown in the following figure.

Figure 10: **Math Sequencer Instruction**



15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0 Instruction BIT #
A mux	B mux		+,*, sat, bypass
Data Select		Store	Instruction Select

Note: a 0x0 instruction ends the math sequence

There are ways to reduce the amount of arithmetic hardware needed at the expense of clock cycles. For example, $y = (a - b)$ would require a two step operation using $b \times -1$ -> b (b is multiplied by −1 and then stored into register b), followed by a second operation of $a + b$ -> b. The results of an arithmetic operation is stored in either the A or B register array, which is not desirable for the next operand input needed to follow the arithmetic operation. To work around these issues, a bypass instruction is necessary to move data from the A to the B register array or from the B to the A register. To expand the computational capabilities for additional algorithms and reduce clock cycles, custom operators can be added to the instruction pipeline during a shift, divide, square root, or another time. For the SPFP PID example using a C++ based Math Sequencer, the only operators needed are saturate, bypass, multiplier, and adder.

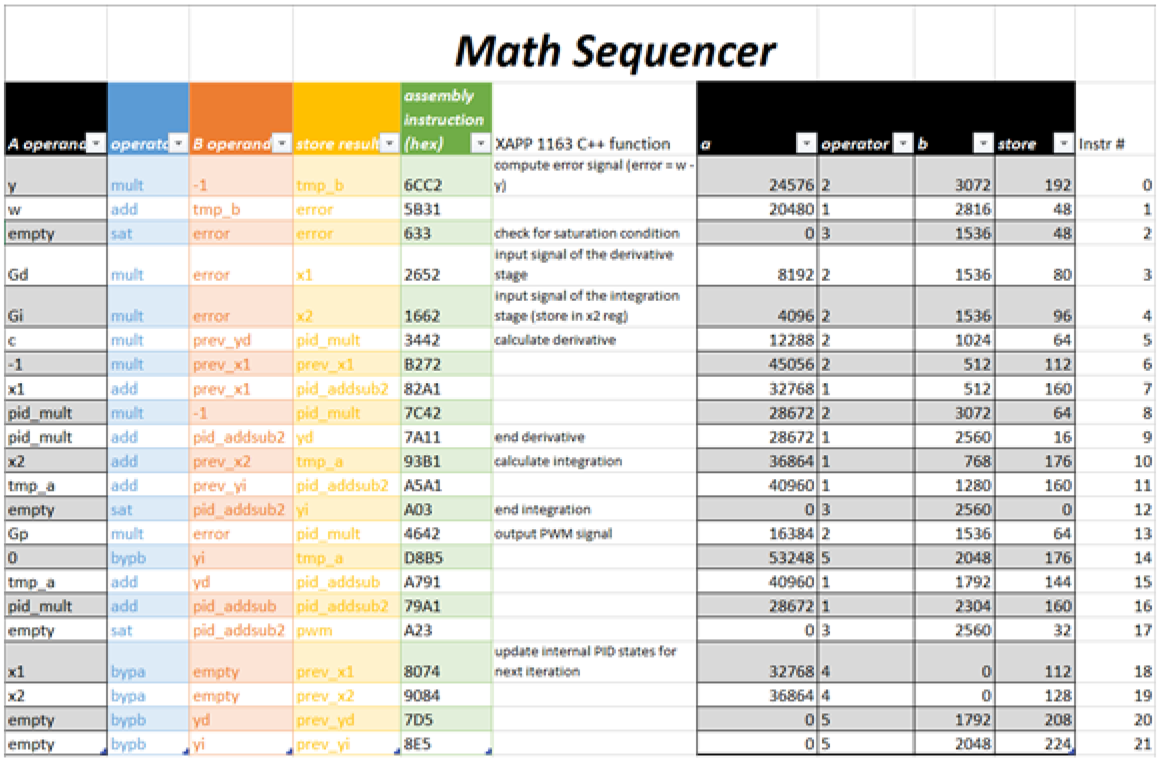
The C++ for the Math Sequencer is very simple and easy to understand when written in C:

```
#include "ms.h"

void ms(float w_in, float y_in, float &pwm) {

    // for register & instruction details see math_sequencer_rv2.xls (MS
Excel Spreadsheet)

    // A mux  data
    static float a_mux[] = {0, Gi, Gd, c, Gp, 0, 0, 0, 0, 0, 0, minus1,
plus1, zeroc};
    // B mux data
    static float b_mux[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, minus1,
plus1, zeroc};
    // constant definitions
#pragma HLS ARRAY_PARTITION variable=b_mux complete dim=1

    // load data from interface
    a_mux[5] = (float) w_in; // cast variable to float
    a_mux[6] = (float) y_in;

    // setup instructions
    unsigned short mnemonics[23] =
{0x6CC2,0x5B31,0x633,0x2652,0x1662,0x3442,0xB272,0x82A1,0x7C42,0x7A11,0x93B1
,0xA5A1,0xA03,0x4642,0xD8B5,0xA791,0x79A1,0xA23,0x8074,0x9084,0x7D5,0x8E5};
    const short num_instr = 22;

    ap_uint<16> instruction;
    ap_uint<4> instr_sel; // instruction mux controls
    ap_uint<4> store;
    ap_uint<4> dsel_b, dsel_a;

    float a_sel_data, b_sel_data; // variables for data management &
results storage
    float op_results; // 32 bit results
    float fsat_o; // float data types needed to support saturate


        instruction_loop : for (short inst_loop = 0; inst_loop < num_instr;
inst_loop++) {
            // split up instruction into tasks
#pragma HLS PIPELINE
            instruction = (ap_uint<16>) mnemonics[inst_loop];

            // split out instructions into sub blocks
            instr_sel = instruction & 0x000F; // instruction select
            store = (instruction & 0x00F0) >> 4; // store results where?
            dsel_b = (instruction & 0x0F00) >> 8; // source B side mux from
what register?
            dsel_a = (instruction & 0xF000) >> 12; // source A side mux
from what register?

            // determine A select
            a_sel_data = a_mux[dsel_a];

            // determine B select
            b_sel_data = b_mux[dsel_b];
            switch(instr_sel) {
                case 0: break; // invalid instruction
                case 1: op_results = a_sel_data + b_sel_data; break; // a+b
                case 2: op_results = a_sel_data * b_sel_data; break; // a*b
                case 3: if((b_sel_data >= min_limit) && (b_sel_data <=
max_limit)) // saturate
                            fsat_o = b_sel_data;
                        else if (b_sel_data < min_limit)
                            fsat_o = min_limit;
                        else if (b_sel_data > max_limit)
                            fsat_o = max_limit;
                        op_results = fsat_o; break;
                case 4: op_results = a_sel_data; break; // bypass a
                case 5: op_results = b_sel_data; break; // bypass b
                // case 6: op_results = a_sel_data / b_sel_data; break; //
a/b
                break;
            } // end instr_sel

            switch(store) {
                case 0: b_mux[8] = op_results; break; // yi
                case 1: b_mux[7] = op_results; break; // yd
                case 2: b_mux[1] = op_results; break; // pwm
                case 3: b_mux[6] = op_results; break; // error
                case 4: a_mux[7] = op_results; break; // pid_mult
                case 5: a_mux[8] = op_results; break; // x1; a_mux
                case 6: a_mux[9] = op_results; break; // x2; a_mux
                case 7: b_mux[2] = op_results; break; // prev_x1
                case 8: b_mux[3] = op_results; break; //prev_x2
                case 9: b_mux[9] = op_results; break; // pid_addsub
                case 10: b_mux[10] = op_results; break; // pid_addsub2
                case 11: a_mux[10] = op_results; break; // tmp_a; a_mux
                case 12: b_mux[11] = op_results; break; // tmp_b
                case 13: b_mux[4] = op_results; break; // prev_yd
                case 14: b_mux[5] = op_results; break; // prev_yi
                case 15: break; // invalid
            } // end store results

        } // end instruction_loop
    pwm = b_mux[1]; // PWM is a pass by reference variable

} // end math sequencer
```

The VMC enables functional development and debug of user-authored C and C++ designs. The traditional methods of C and C++ debugging (GNU debugger or Microsoft Visual C) and `printf` options are available in addition to Simulink scopes, display, data logging, MATLAB scripts, and others that are natural for DSP development in a MATLAB Simulink development environment.

To simplify the microcode instruction creation, the following Microsoft Excel spreadsheet was used to augment the generation of the hexadecimal values needed for a PID sequence.

Figure 11: **Excel Spreadsheet Used to Generate Math Sequencer Instruction Sequence**

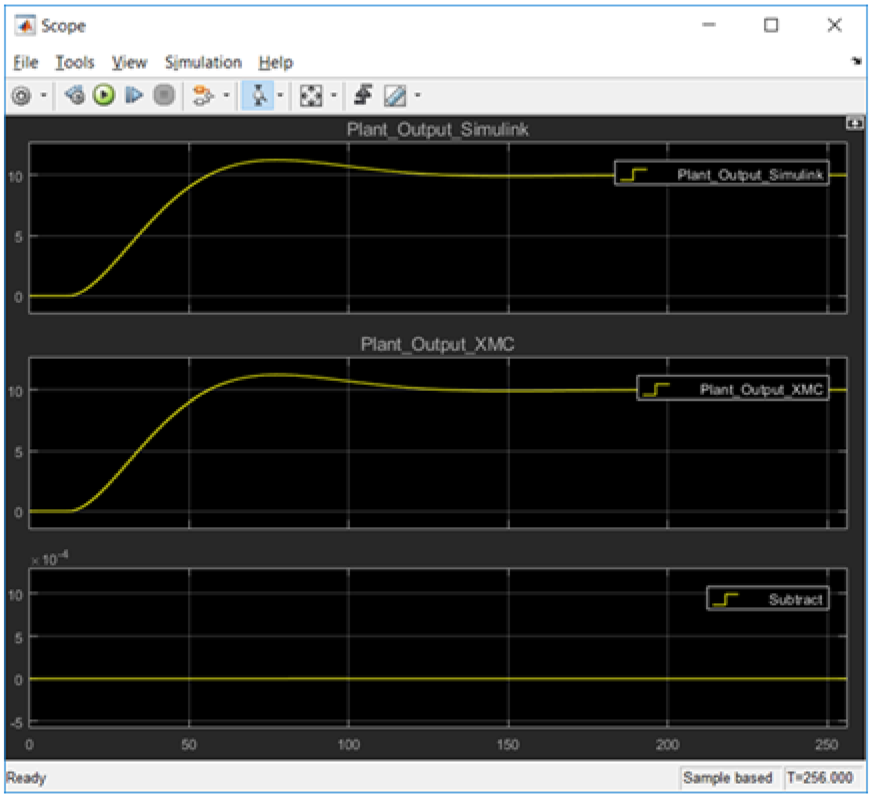


Math Sequencer

A operan	operat	B operand	store resul	assembly instruction (hex)	XAPP 1163 C++ function	a	operator	b	store	Instr #
y	mult	-1	tmp_b	6CC2	compute error signal (error = w - y)	24576	2	3072	192	0
w	add	tmp_b	error	5B31		20480	1	2816	48	1
empty	sat	error	error	633	check for saturation condition	0	3	1536	48	2
Gd	mult	error	x1	2652	input signal of the derivative stage	8192	2	1536	80	3
Gi	mult	error	x2	1662	input signal of the integration stage (store in x2 reg)	4096	2	1536	96	4
c	mult	prev_yd	pid_mult	3442	calculate derivative	12288	2	1024	64	5
-1	mult	prev_x1	prev_x1	B272		45056	2	512	112	6
x1	add	prev_x1	pid_addsub2	82A1		32768	1	512	160	7
pid_mult	mult	-1	pid_mult	7C42		28672	2	3072	64	8
pid_mult	add	pid_addsub2	yd	7A11	end derivative	28672	1	2560	16	9
x2	add	prev_x2	tmp_a	93B1	calculate integration	36864	1	768	176	10
tmp_a	add	prev_yi	pid_addsub2	A5A1		40960	1	1280	160	11
empty	sat	pid_addsub2	yi	A03	end integration	0	3	2560	0	12
Gp	mult	error	pid_mult	4642	output PWM signal	16384	2	1536	64	13
0	bypb	yi	tmp_a	D8B5		53248	5	2048	176	14
tmp_a	add	yd	pid_addsub	A791		40960	1	1792	144	15
pid_mult	add	pid_addsub	pid_addsub2	79A1		28672	1	2304	160	16
empty	sat	pid_addsub2	pwm	A23		0	3	2560	32	17
x1	bypa	empty	prev_x1	8074	update internal PID states for next iteration	32768	4	0	112	18
x2	bypa	empty	prev_x2	9084		36864	4	0	128	19
empty	bypb	yd	prev_yd	7D5		0	5	1792	208	20
empty	bypb	yi	prev_yi	8E5		0	5	2048	224	21

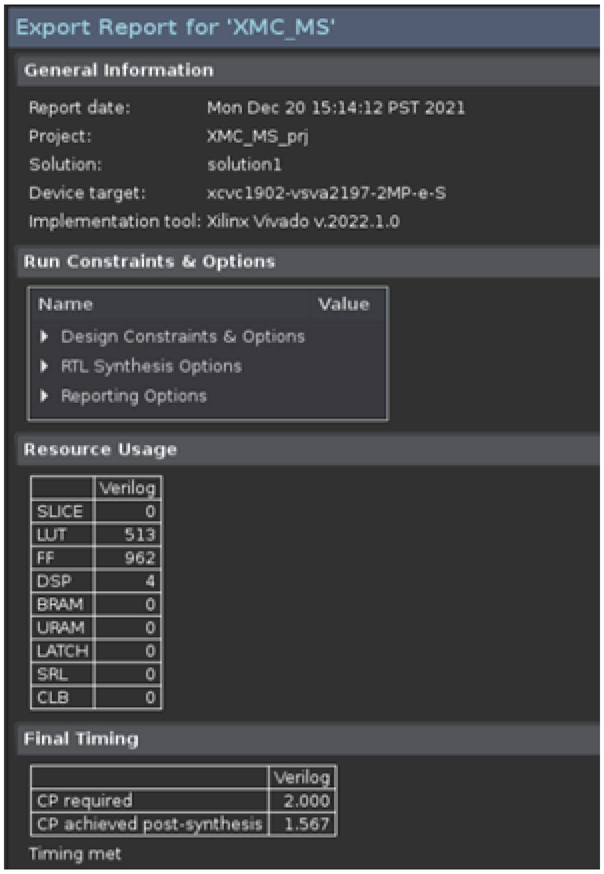
Once again, functional simulation indicates there are no functional or dynamic range differences found between the Simulink golden reference model and the Math Sequencer.

Figure 12: **Simulink versus Math Sequencer Simulation Results**



As demonstrated in the following figure, by applying two HLS #pragma instructions to the C++ code, the following PL implemented results in the Clock Rate = 1/1.567e−9 = 638.2 MHz clock; Sample Rate = 1/(83 × 1.567e−9) = 7.7 MSPS.

Figure 13: **PL Implemented Results**



Export Report for 'XMC_MS'

General Information

Report date:	Mon Dec 20 15:14:12 PST 2021
Project:	XMC_MS_prj
Solution:	solution1
Device target:	xcvc1902-vsva2197-2MP-e-S
Implementation tool:	Xilinx Vivado v.2022.1.0

Run Constraints & Options

Name	Value
▸ Design Constraints & Options	
▸ RTL Synthesis Options	
▸ Reporting Options	

Resource Usage

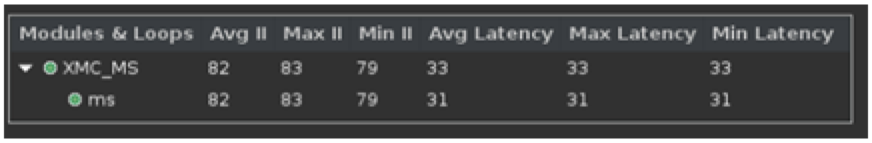
	Verilog
SLICE	0
LUT	513
FF	962
DSP	4
BRAM	0
URAM	0
LATCH	0
SRL	0
CLB	0

Final Timing

	Verilog
CP required	2.000
CP achieved post-synthesis	1.567

Timing met

Figure 14: **Math Sequencer Implementation Results**



Modules & Loops	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
▾ ● XMC_MS	82	83	79	33	33	33
● ms	82	83	79	31	31	31

The advantage of a Math Sequencer is that it can easily change the operators based on the algorithm requirements. For example, a division operator or a square root function can be added to calculate the magnitude. Also, the Math Sequencer can change the instruction sequence at a later date without changing the hardware implementation. Further, the Math Sequencer can reduce resources at the expense of latency. Direct comparison of the HLS Toolbox and C++ Math Sequencer PID control loop implementations is demonstrated in the following table.

Table 2: **Resources, Latency, Clock Frequency, and Sample Rate Comparison**

	DSP	LUTs	FFs	Block RAM	Latency (Clocks)	Clock (MHz)	Sample Rate (MSPS)
VMC HLS Toolbox	5	565	505	0	69	472	6.8
C++ Math Sequencer	4	513	962	0	83	638.2	7.7

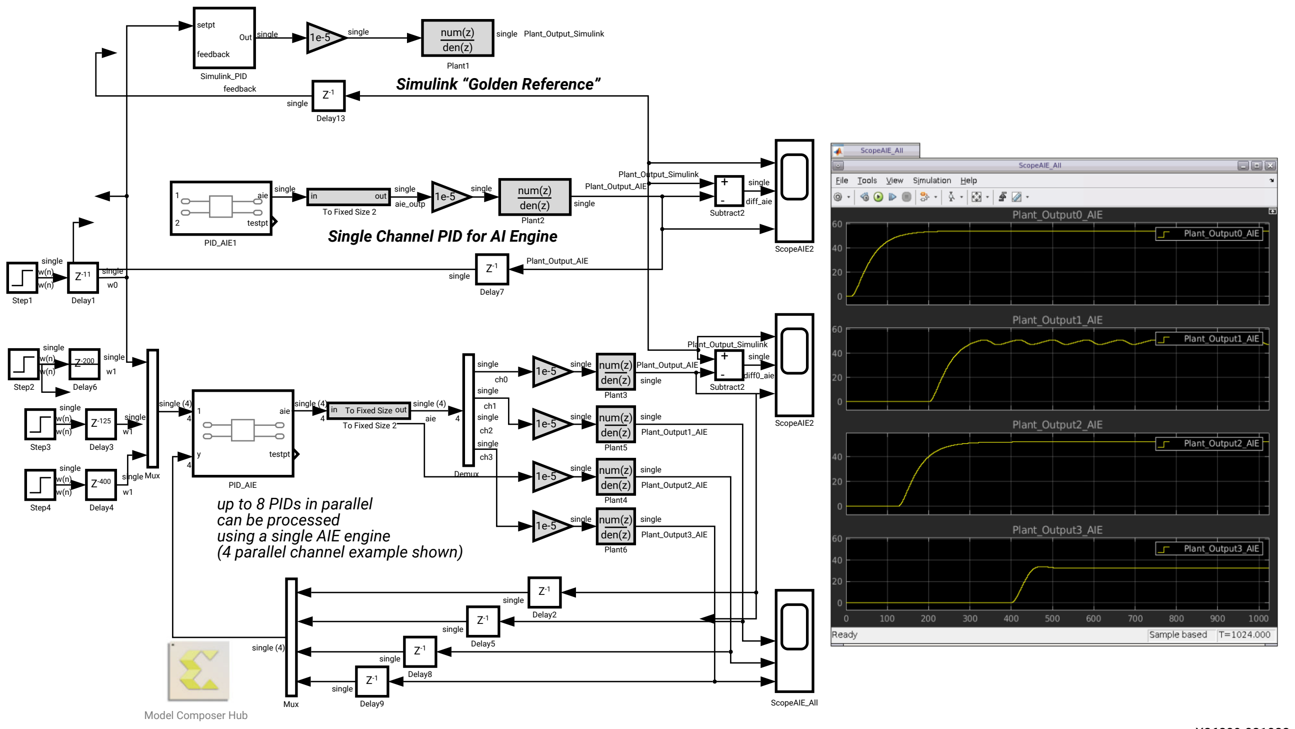
VMC AI Engine Design

AI Engines add another flexible dimension to numerical computations. In order to show the versatility of the Versal AI Engine, the PID was re-written to target an AI Engine. A single channel SPFP AI Engine based PID intrinsic source code is shown in the following code.

```
// error = setpt - feedback
error = upd_elem(error, 0, readincr(setpt)); // MM 1/6 was inp_data, 0, 
readincr...
scratch_pad = upd_elem(scratch_pad, 0, readincr(feedback));
error = fpsub(error, scratch_pad); // save error data

// proportional code
acc = fpmul(error, *Gp_ptr); // acc now holds proportional path results

writeincr(testpt, ext_elem(error, 0)); // MM

// derivative code
inp_data = fpmul(error, *Gd_ptr); // X1(n)
scratch_pad = fpsub( inp_data, fpmul(derivative_delay1, *C_ptr) ); // X1(n)-
CYd(n-1)
scratch_pad = fpsub(scratch_pad, derivative_delay); // Yd(n) = X1(n)-
CYd(n-1)-X1(n-1)
derivative_delay = inp_data;
derivative_delay1 = scratch_pad;

// add proportional & derivative results
acc = fpadd(acc, scratch_pad);

// integral code
inp_data = fpmul(error, *Gi_ptr); // X2(n)
scratch_pad = fpadd(inp_data, integral_delay);
integral_delay = inp_data;
scratch_pad = fpadd(integral_delay1, scratch_pad);

// test for saturation for integral path (ie: prevent integral anti-windup)
if (ext_elem(scratch_pad,0) > max_clip )
    scratch_pad = upd_elem(scratch_pad, 0, max_clip);
else if (ext_elem(scratch_pad,0) < min_clip )
    scratch_pad = upd_elem(scratch_pad, 0, min_clip);

integral_delay1 = scratch_pad;

// add proportional, integral, derivative results
acc = fpadd(acc, scratch_pad);

// test for saturation
if (ext_elem(acc,0) > max_clip)
    acc = upd_elem(acc, 0, max_clip);
else if (ext_elem(acc,0) < min_clip )
    acc = upd_elem(acc, 0, min_clip);

// write out results for servo lane 0
writeincr(outp, ext_elem(acc, 0));

}
```

A single channel PID implementation only utilizes one -eighth of the full AI Engine capacity. Alternately, the vector processor's single instruction multiple data (SIMD) capability can be used to process between one and eight PIDs in parallel. The following figure is an example of both a single channel (reference `PID.cc` source code) and four channel (reference `PID_rv2.cc` source code) SPFP PIDs running concurrently on an AI Engine (reference source code: `PID_rv2.cc`).

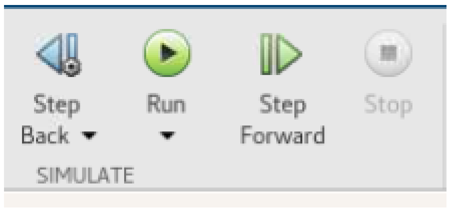
Figure 15: **Four Channel AI Engine PID Single Channel PID Compared to a Single Channel AI Engine PID and the Simulink Golden Reference (Reference Design: `ClosedLoopPID_ACAP_rv2.slx`)**



X26289-021022

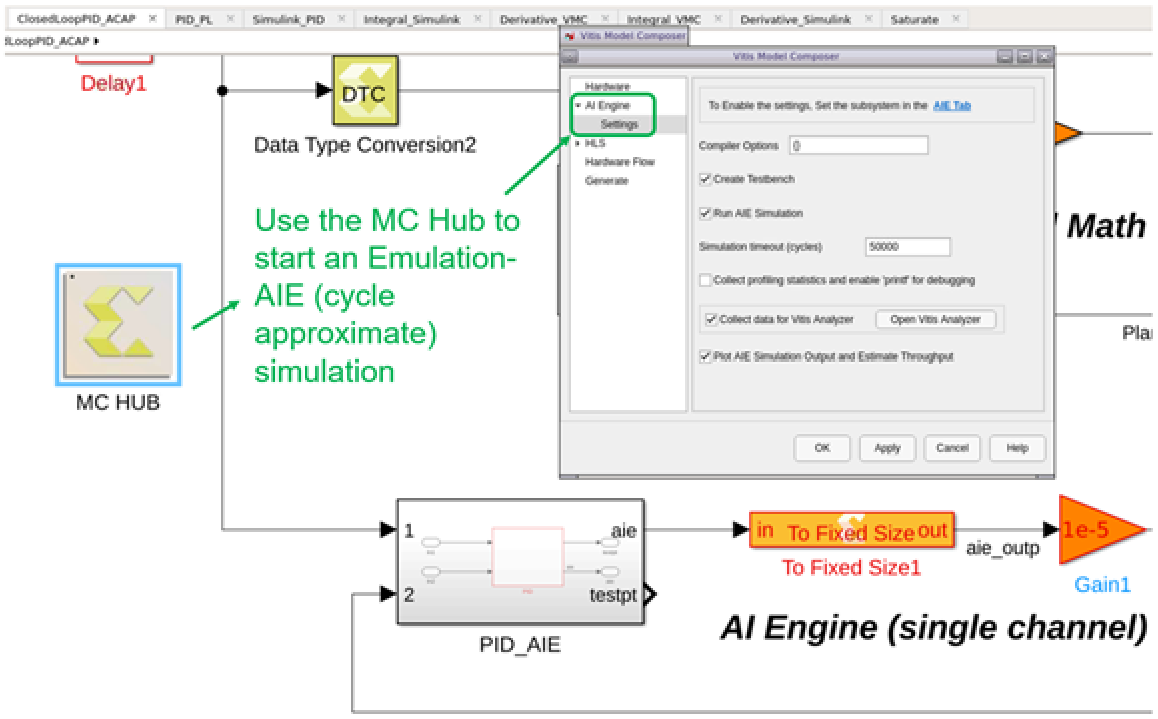
The four channel scope results (ScopeAIE_All) display the results for four different sets of Kp, Ki, and Kd coefficients. The C++ for the AI Engine was functionally debugged during development via Vitis Emulation-SW simulations executed by pushing the Simulink run button.

Figure 16: **Functionally Simulating (Vitis Emulation-SW Simulation) an AI Engine Design in Simulink**



Functional debugging (Vitis Emulation-SW simulation) can be one to two orders of magnitude faster than running the same models using cycle approximate simulations (Vitis Emulation-AI Engine simulations). Therefore, a large part of development should use functional simulations in order to reduce development time and simplify debug of any new design. After functional verification of the PID controller completes, the Vitis Emulation-AI Engine (cycle approximate) simulator is used via the MC Hub token as demonstrated in the following figure.

Figure 17: **Running the Bit Accurate and Cycle Approximate (Emulation-AI Engine) Simulation**



To Enable the settings, Set the subsystem in the AIE Tab

Compiler Options: {}

[✓] Create Testbench

[✓] Run AIE Simulation

Simulation timeout (cycles): 50000

[ ] Collect profiling statistics and enable 'printf' for debugging

[✓] Collect data for Vitis Analyzer

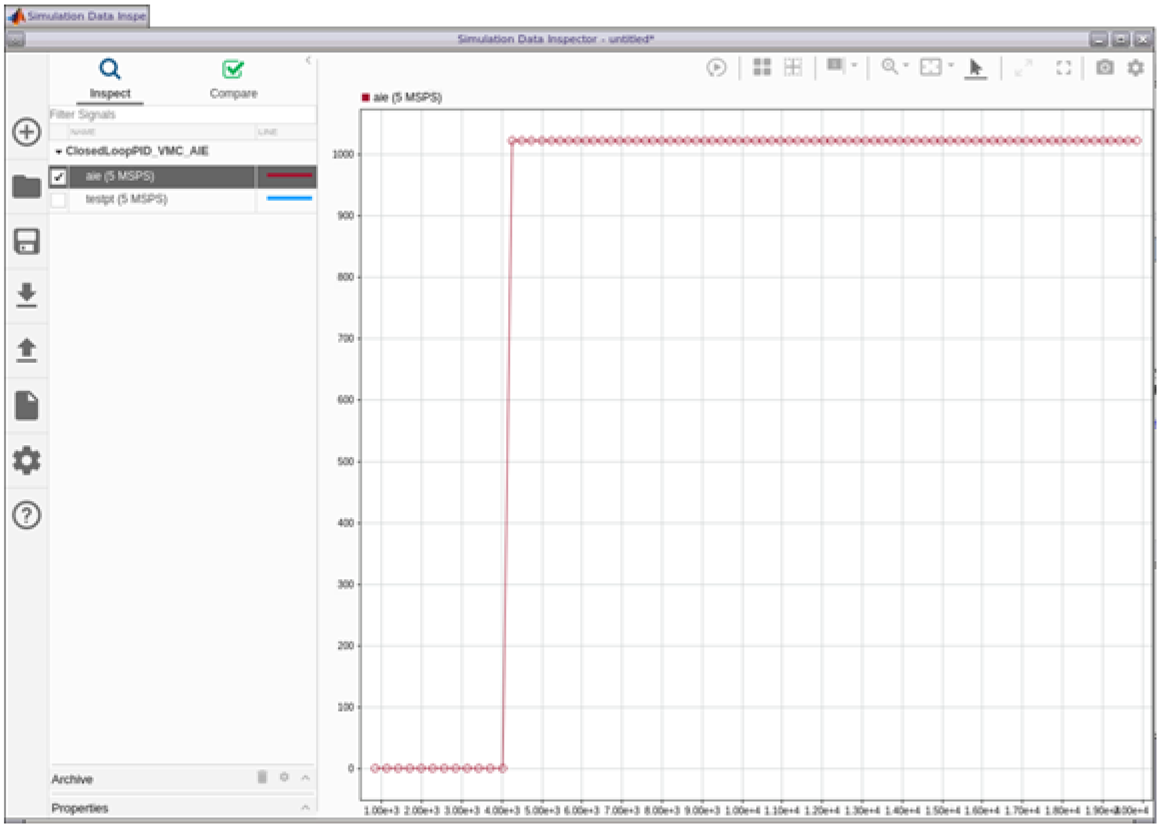
[✓] Plot AIE Simulation Output and Estimate Throughput

Cycle approximate simulations allow improved throughput by changing the source code or applying compiler directives and debugging potential cycle accurate implementation issues. When the Model Composer Hub is used for cycle approximate simulation, the following automated steps are performed:

1. A test bench using the Simulink design is created, and adaptive dataflow graph (ADF) is generated.
2. The Emulation-AI Engine Vitis flow is run using the Vitis tools.
3. The Vitis analyzer opens for detailed analysis.
4. The Emulation-AI Engine simulation output is plotted and estimates the throughput.

Plotting the cycle approximate (Emulation-AI Engine simulation) output estimates for the single-channel AI Engine based PID design has a 5 MSPS throughput as shown in the following figure.

Figure 18: **Emulation-AI Engine Throughput Estimates**

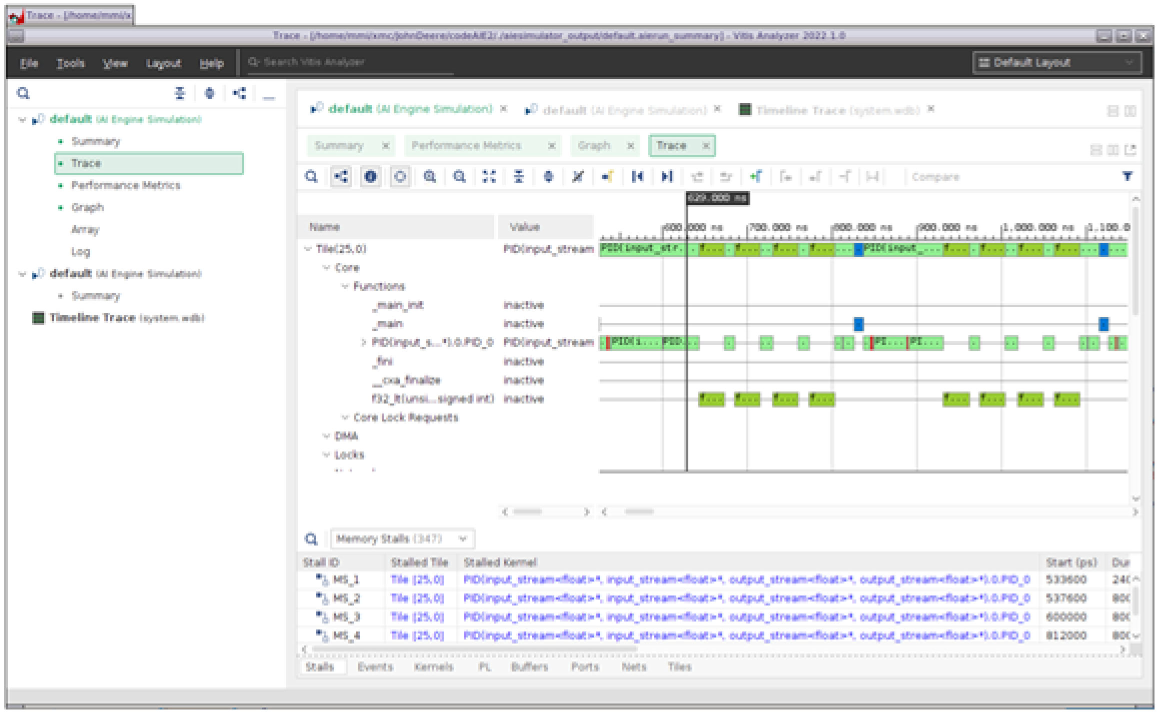


The four-channel AI Engine PID has a 4 MSPS throughput. The difference in sample rate performance between the single-channel and four-channel PID is the conditional statements necessary to iterate across four parallel channels. Line 105 in `PID_rv2.cc` has a constant `num_pids` which defines the PIDs for loop lengths. The existing value is four, but the maximum value is eight. For the sake of simplicity, and to keep the Simulink ADF sheet from being too cluttered for explanation purposes, an arbitrary four channels was chosen.

Emulation AI Engine Analysis

VMC gives you the ability to explore Vitis based performance metrics, graph, array, trace, and log details using the Vitis Analyzer GUI.

Figure 19: **Vitis Analyzer Results Analysis**



Trace - [/home/mmi/xmc/johnDeere/codeAIE2/./aiesimulator_output/default.aierun_summary] - Vitis Analyzer 2022.1.0

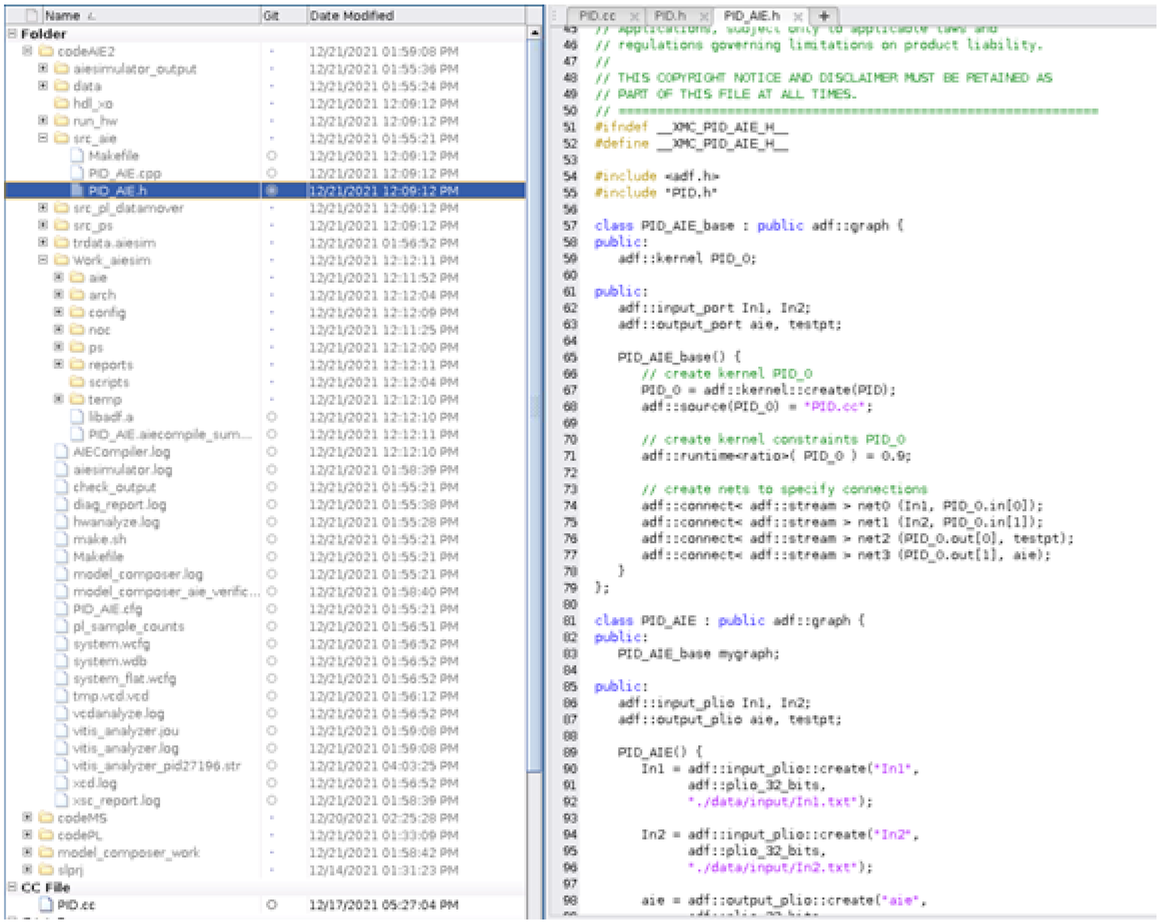
Name	Value
Tile(25,0)	PID(input_stream
Core	
Functions	
_main_init	inactive
_main	inactive
PID(input_s...*).0.PID_0	PID(input_stream
_fini	inactive
__cxa_finalize	inactive
f32_lt(unsi...signed int)	inactive
Core Lock Requests	
DMA	
Locks	

Memory Stalls (347)

Stall ID	Stalled Tile	Stalled Kernel	Start (ps)	Dur
MS_1	Tile [25,0]	PID(input_stream<float>*, input_stream<float>*, output_stream<float>*, output_stream<float>*).0.PID_0	533600	24(
MS_2	Tile [25,0]	PID(input_stream<float>*, input_stream<float>*, output_stream<float>*, output_stream<float>*).0.PID_0	537600	80(
MS_3	Tile [25,0]	PID(input_stream<float>*, input_stream<float>*, output_stream<float>*, output_stream<float>*).0.PID_0	600000	80(
MS_4	Tile [25,0]	PID(input_stream<float>*, input_stream<float>*, output_stream<float>*, output_stream<float>*).0.PID_0	812000	80(

All Vitis files generated from a VMC Emulation-AI Engine simulation are available for exploration or reuse as shown in the following figure.

Figure 20: **Vitis Emulation-SW and Emulation-AI Engine Directories**



Name	Git	Date Modified
Folder		
codeAIE2	-	12/21/2021 01:59:08 PM
aiesimulator_output	-	12/21/2021 01:55:36 PM
data	-	12/21/2021 01:55:24 PM
hdl_xo	-	12/21/2021 12:09:12 PM
run_hw	-	12/21/2021 12:09:12 PM
src_aie	-	12/21/2021 01:55:21 PM
Makefile	○	12/21/2021 12:09:12 PM
PID_AIE.cpp	○	12/21/2021 12:09:12 PM
PID_AIE.h	○	12/21/2021 12:09:12 PM
src_pl_datamover	-	12/21/2021 12:09:12 PM
src_ps	-	12/21/2021 12:09:12 PM
trdata.aiesim	-	12/21/2021 01:56:52 PM
Work_aiesim	-	12/21/2021 12:12:11 PM
aie	-	12/21/2021 12:11:52 PM
arch	-	12/21/2021 12:12:04 PM
config	-	12/21/2021 12:12:09 PM
noc	-	12/21/2021 12:11:25 PM
ps	-	12/21/2021 12:12:00 PM
reports	-	12/21/2021 12:12:11 PM
scripts	-	12/21/2021 12:12:04 PM
temp	-	12/21/2021 12:12:10 PM
libadf.a	○	12/21/2021 12:12:10 PM
PID_AIE.aiecompile_sum...	○	12/21/2021 12:12:11 PM
AIECompiler.log	○	12/21/2021 12:12:10 PM
aiesimulator.log	○	12/21/2021 01:58:39 PM
check_output	○	12/21/2021 01:55:21 PM
diag_report.log	○	12/21/2021 01:55:38 PM
hwanalyze.log	○	12/21/2021 01:55:28 PM
make.sh	○	12/21/2021 01:55:21 PM
Makefile	○	12/21/2021 01:55:21 PM
model_composer.log	○	12/21/2021 01:55:21 PM
model_composer_aie_verific...	○	12/21/2021 01:58:40 PM
PID_AIE.cfg	○	12/21/2021 01:55:21 PM
pl_sample_counts	○	12/21/2021 01:56:51 PM
system.wcfg	○	12/21/2021 01:56:52 PM
system.wdb	○	12/21/2021 01:56:52 PM
system_flat.wcfg	○	12/21/2021 01:56:52 PM
tmp.vcd.vcd	○	12/21/2021 01:56:12 PM
vcdanalyze.log	○	12/21/2021 01:56:52 PM
vitis_analyzer.jou	○	12/21/2021 01:59:08 PM
vitis_analyzer.log	○	12/21/2021 01:59:08 PM
vitis_analyzer_pid27196.str	○	12/21/2021 04:03:25 PM
xcd.log	○	12/21/2021 01:56:52 PM
xsc_report.log	○	12/21/2021 01:58:39 PM
codeMS	-	12/20/2021 02:25:28 PM
codePL	-	12/21/2021 01:33:09 PM
model_composer_work	-	12/21/2021 01:58:42 PM
slprj	-	12/14/2021 01:31:23 PM
CC File		
PID.cc	○	12/17/2021 05:27:04 PM

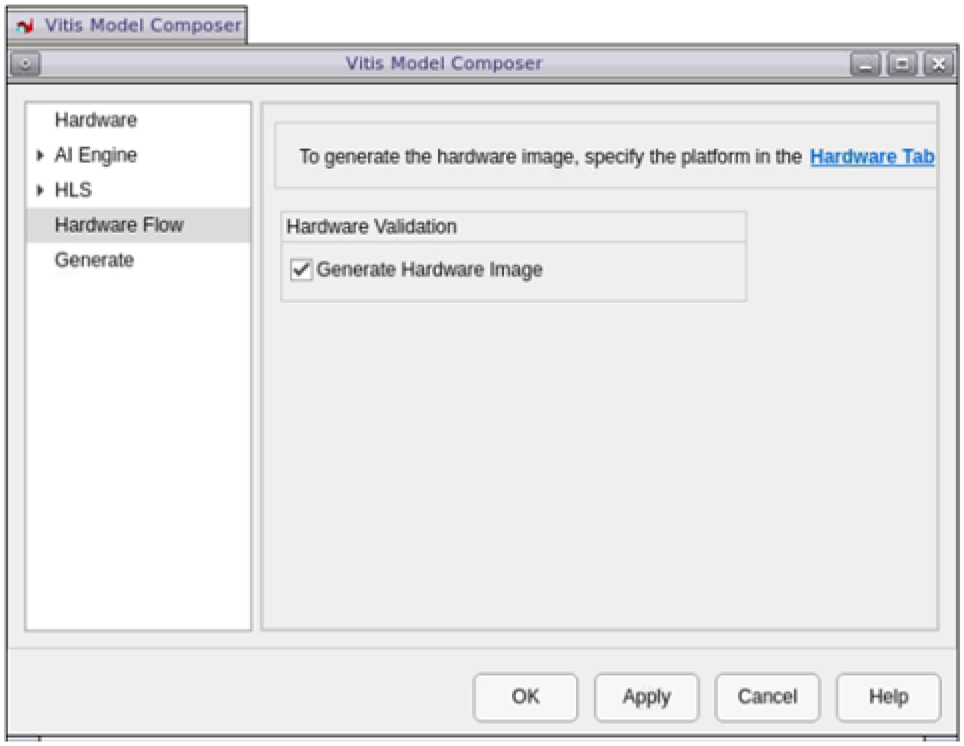
```
// applications, subject only to applicable laws and
// regulations governing limitations on product liability.
//
// THIS COPYRIGHT NOTICE AND DISCLAIMER MUST BE RETAINED AS
// PART OF THIS FILE AT ALL TIMES.
// ====================================================================
#ifndef __XMC_PID_AIE_H__
#define __XMC_PID_AIE_H__

#include <adf.h>
#include "PID.h"

class PID_AIE_base : public adf::graph {
public:
   adf::kernel PID_0;

public:
   adf::input_port In1, In2;
   adf::output_port aie, testpt;

   PID_AIE_base() {
      // create kernel PID_0
      PID_0 = adf::kernel::create(PID);
      adf::source(PID_0) = "PID.cc";

      // create kernel constraints PID_0
      adf::runtime<ratio>( PID_0 ) = 0.9;

      // create nets to specify connections
      adf::connect< adf::stream > net0 (In1, PID_0.in[0]);
      adf::connect< adf::stream > net1 (In2, PID_0.in[1]);
      adf::connect< adf::stream > net2 (PID_0.out[0], testpt);
      adf::connect< adf::stream > net3 (PID_0.out[1], aie);
   }
};

class PID_AIE : public adf::graph {
public:
   PID_AIE_base mygraph;

public:
   adf::input_plio In1, In2;
   adf::output_plio aie, testpt;

   PID_AIE() {
      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");

      aie = adf::output_plio::create("aie",
```

A makefile is generated as part of the Emulation-AI Engine simulation in addition to a test bench, AI Engine Adaptive DataFlow graph files (`PID_AIE.cpp`, `PID_AIE.h`), and user defined source files. You can execute the VMC created `./code/Work_aiesim/Makefile` in order to perform a Vitis Emulation-SW (make all_x86) or a Vitis Emulation-AI Engine (through a Vitis xterm make all) standalone simulation.

AI Engine Hardware Validation

AI Engine, AI Engine+RTL, or AI Engine+HLS based designs can be run in the hardware using the Model Composer Hub through the Generate Hardware Image option shown in the following figure.

Figure 21: **Generate Hardware Image Option in the VMC Hub**



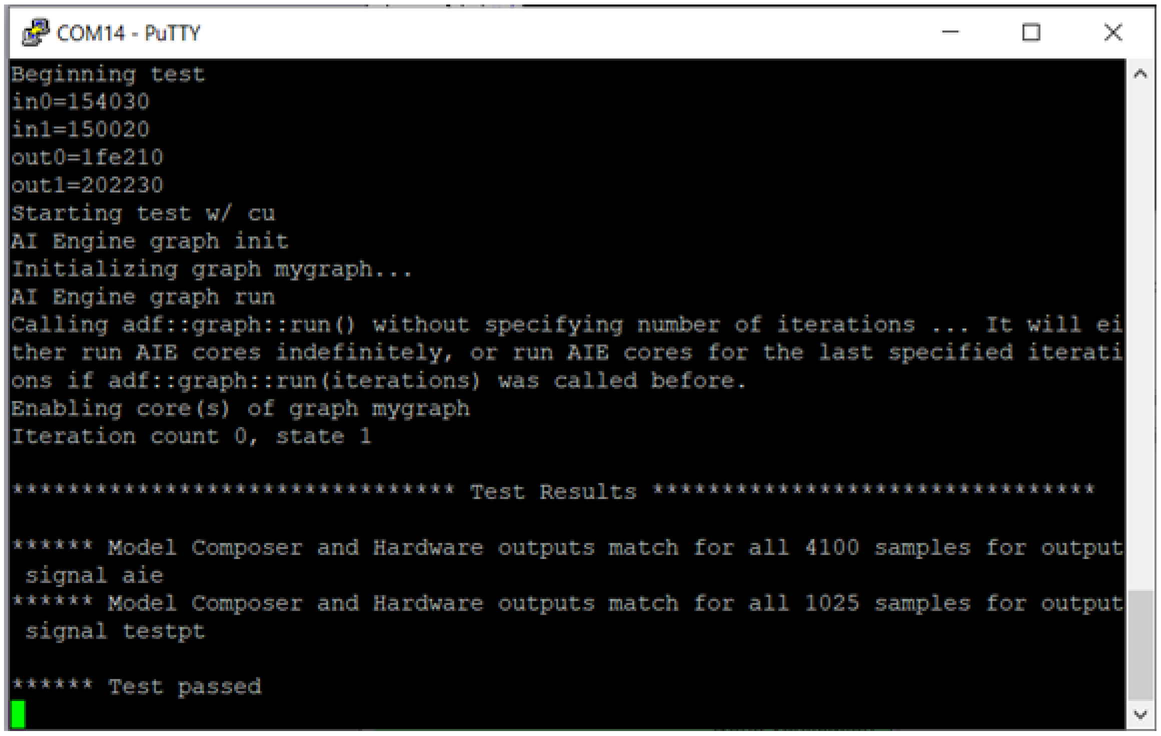
To generate the hardware image, specify the platform in the Hardware Tab

Hardware Validation

✓ Generate Hardware Image

The following figure demonstrates the hardware validation flow as an automated method to verify a working AI Engine hardware-based design where the com port echos the pass/fail sample based results for the four-channel AI Engine design.

Figure 22: **Com Port Results from Hardware Validation**



```
Beginning test
in0=154030
in1=150020
out0=1fe210
out1=202230
Starting test w/ cu
AI Engine graph init
Initializing graph mygraph...
AI Engine graph run
Calling adf::graph::run() without specifying number of iterations ... It will ei
ther run AIE cores indefinitely, or run AIE cores for the last specified iterati
ons if adf::graph::run(iterations) was called before.
Enabling core(s) of graph mygraph
Iteration count 0, state 1

******************************** Test Results ********************************

****** Model Composer and Hardware outputs match for all 4100 samples for output
 signal aie
****** Model Composer and Hardware outputs match for all 1025 samples for output
 signal testpt

****** Test passed
```

To simply change the test bench rather than going through the longer process of recompiling the hardware design, perform the following sequence of events:

1. Change the test bench.
2. Use the VMC Hub to re-simulate.

Being able to switch out the test bench without recompiling the hardware can significantly reduce validation for larger simulation vector sets needed to fully vet a design.

Conclusion

Using a MATLAB Simulink development environment can greatly simplify the effort required to design, debug, and analyze a DSP based Versal ACAP design. In this application note, a PID controller was simulated and implemented in PL using the VMC HLS Toolbox and a custom C++ based Math sequencer in PL, targeting the Versal DSP58 single precision floating point hard macros. The AI Engine implementation demonstrated how the single instruction multiple data (SIMD) vector processor can be harnessed to perform up to eight PIDs concurrently. The resource comparison for a one PID loop VMC HLS Toolbox, Math Sequencer implementation, and four parallel PID loops running concurrently on one AI Engine is shown in the following table.

Table 3: **Resources, Latency, Clock Frequency and Sample Rate Comparison for a Single Precision Floating Point PID Implementation**

	DSP	LUTs	FFs	Block RAM	AI Engine	Latency (Clocks)	Clock (MHz)	Sample Rate (MSPS)
VMC native blocks (single channel)	5	565	505	0	0	69	472	6.8
Math Sequencer (single channel)	4	513	962	0	0	83	684.3	7.7
AI Engine (4 channel)	0	0	0	0	1		1 GHz	4

For a single copy of the VMC HLS Toolbox PID, five DSP, 565 LUT, 505 FF would be consumed, which is an almost negligible number of gates in a Versal device. For one or two PIDs with available gates, using dedicated gates is the best approach. To run a more complicated algorithm such as field oriented control (FOC) application, more dedicated hardware with more math operators is needed, which will increase costs and resources.

For eight PID loops using the VMC HLS toolbox approach, we need $8 \times (565\ \mathrm{LUT} + 505\ \mathrm{FF} + 5\ \mathrm{DSP58}) = (4520\ \mathrm{LUT} + 4040\ \mathrm{FF} + 40\ \mathrm{DSP58})$, which will all independently run at 6.84 MSPS. But 6.84 MSPS is significantly faster than what a brushless DC motor with ~40 KHz loop bandwidth might require. In other words, with a 40 KHz PID loop bandwidth, you process one sample every 40 KHz or 40 KSPS. If the VMC HLS toolbox PID is $6.84\ \mathrm{MSPS}/40\ \mathrm{KSPS} = 171$ times faster than required for a brushless DC motor controller, and you want to drive down cost, you should resource share the multiplication, adder, and subtract operators over time. The Math Sequencer is explicitly designed to be a low cost way to sequentially process arithmetic operations over time.

The Math Sequencer runs a single PID loop at 7.7 MSPS. The register set of the Math Sequencer can be enhanced and sequentially process eight PIDs using a single math sequencer, but then the achievable sample rate per channel will drop linearly to ~12.5% of a single channel throughput. 7.7 MSPS / 8 = 0.96 MSPS which is still ~24x faster than needed for a single 40 KSPS brushless DC motor control loop, and eight copies of dedicated hardware are not needed. A more complex algorithm like a FOC loop using a Math Sequencer can be processed, and the programming would be complex, but there are advantages to a Math Sequencer. Not only are they inexpensive, they only need to be programmed once, no micro controller licensing is required, and no support tools are required to buy and learn. Any arithmetic algorithm can be customized, and the controller field can be updated without having to change the design, much like a processor.

When considering the AI Engines, which have both a 32-bit RISC processor and a vector processor, the vector processor SIMD performs using the same arithmetic operator across all lanes. There are eight parallel lanes for floating point operations. Each individual lane is used to perform the same: four multiplications, three subtracts, and four adds for eight independent PIDs using one AI Engine. If the average brushless DC motor control loop runs at 40 KSPS, the PID loops are running roughly 100x faster than required. At 40 KSPS, an AI Engine would be idle ~99% of the time because it simply does not have enough work to do. This gives you the ability to consider more complex algorithms like field oriented control (FOC), which has several desirable advantages for up to eight motors running simultaneously using a single AI Engine.

Throughout the process, it is worth noting:

- VMC simplifies test bench development, verification, validation, and debug by utilizing the many inherent Simulink capabilities and toolboxes.
- Using functional simulations to debug and develop a design is significantly faster than using cycle approximate bit accurate simulations.
- Both the test bench and the user design can be created, evaluated, and exported for use with Vitis HLS and Vitis.

References

These documents provide supplemental material useful with this application note:

1. *Floating-Point PID Controller Design with Vivado HLS and System Generator for DSP* (XAPP1163)
2. *Versal AI Core Series Data Sheet: DC and AC Switching Characteristics* (DS957)

Reference Design

Download the reference design files for this application note from the Xilinx website.

Reference Design Matrix

The following checklist indicates the procedures used for the provided reference design.

Table 4: **Reference Design Matrix**

Parameter	Description
General	
Developer name	Mike Mitchell
Target devices	Versal ACAPs
Source code provided?	Y
Source code format (if provided)	Various
Design uses code or IP from existing reference design, application note, 3rd party or Vivado software? If yes, list.	Partial reuse from *PID Controller Design with Model Composer* (XAPP1341)
Simulation	
Functional simulation performed	Y
Timing simulation performed?	Y
Test bench provided for functional and timing simulation?	Y
Test bench format	Model Composer
Simulator software and version	2022.1 Xilinx Tools
SPICE/IBIS simulations	N
Implementation	
Synthesis software tools/versions used	Vivado synthesis
Implementation software tool(s) and version	Vivado Implementation
Static timing analysis performed?	N
Hardware Verification	
Hardware verified?	Y
Platform used for verification	VCK190

The reference design includes the following files:

- `ClosedLoopPID_ACAP.slx`: Single precision floating point PID controller using native VMC blocks, C++ based Math Sequencer, single channel AI Engine designs
- `ClosedLoopPID_ACAP_rv2.slx`: Single precision floating point PID controller example for both a single channel and a four channel AI Engine design
- `ms.cpp`, `ms.h`: PL-based Math Sequencer C++ source files
- `PID.cc`, `PID_rv2.cc`, `PID.h`: AI Engine C++ source files
- `create_library.m`: MATLAB file used to create the C++ simulatable Math Sequencer library block for use in a Simulink design

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
03/09/2022 Version 1.0	
Initial release.	N/A

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright