WP362 (v1.0) March 31, 2010

Repeatable Results with Design Preservation

By: Kate Kelley

Increasingly, FPGA designs are no longer just the "glue logic" of the past; they are becoming more complex every year, often incorporating challenging IP such as PCI Express® cores. The complex modules in newer designs, even when not changing, can present difficulties when attempting to meet quality-of-result (QoR) requirements. Time spent trying to maintain timing in these modules is not only frustrating, but often unproductive as well.

The *design preservation flow* solves this issue by allowing the customer to meet timing on the critical module(s) of the design and then reuse the implementation results in future iterations. This reduces the number of implementation iterations in the timing closure phase of the design.

© Copyright 2010 Xilinx, Inc. XILINX, the Xilinx logo, Virtex, Spartan, ISE, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license. All other trademarks are the property of their respective owners.

Benefits of Design Preservation

The main benefit of design preservation is having repeatable results after each design change to reduce the number of implementation runs during the timing closure phase. After timing on a portion of the design is met, the implementation results (placement and routing) are used in the next iteration. This prevents portions of the design that met timing previously from failing timing in the current run.

A second benefit of design preservation is the reduction of time spent in the verification phase. Because the same implementation is used, it is not necessary to do a full verification on modules that have not changed.

Reducing the implementation run time is not a primary goal, but it can often be a secondary benefit. The implementation run time changes for each design run and depends on which modules are being implemented. If the changed module has very tight timing requirements, run time might be longer. On the other hand, if the changed module can meet timing easily and the critical paths are in the preserved modules, run time might be shorter.

How Design Preservation Works

How customers use design preservation is highly design dependent. Some designs, for example, might have one or two critical modules, while others might have four, five, or even six distinct design areas.

For example, assume a design has four areas that can be divided into sections. These areas are called the blue, green, orange, and red modules. Modules that have no design changes can reuse the implementation results in the next iteration. In Figure 1, the blue, green, and orange modules are unchanged; therefore, their placement and routing are imported into the new implementation, while the red module is implemented using the remaining resources. This guarantees that the blue, green, and orange modules have the same timing results.

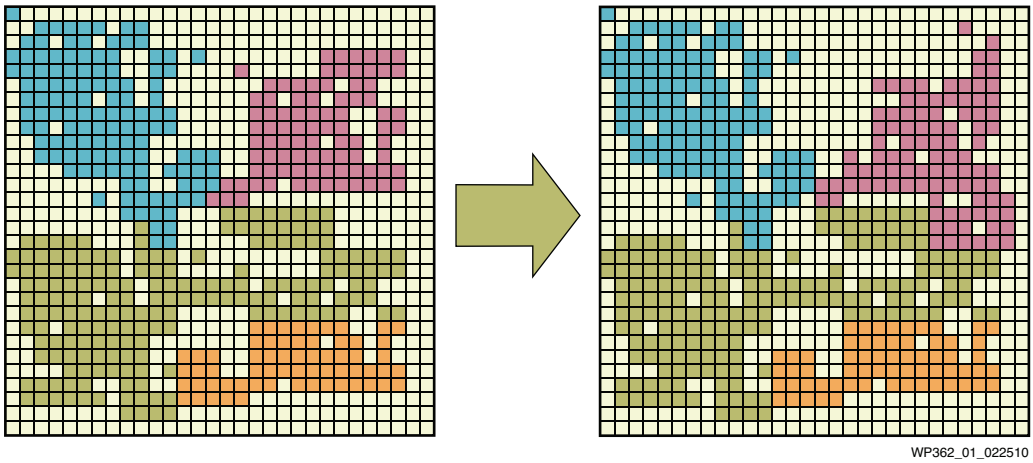

Figure 1: **Design Preservation Example**

While this example has four modules, the design preservation flow can work very well even if only one critical module (like PCIe) is preserved, while the rest of the design is always implemented.

Defining the Design Hierarchy

Although the main benefit of design preservation is reducing iterations during the timing closure phase, it is the hierarchy defined in the RTL design phase that determines whether the design can take advantage of this benefit. Partitions comprise the underlying technology in the design preservation flow, and follow the logical hierarchy of an HDL design. Therefore, the design preservation flow works best with designs that follow good hierarchical rules. Even if the design preservation flow is not taken into consideration during the RTL design phase, there are often independent cores that can still take advantage of this flow.

Determining which modules or entities should comprise a partition is very important for achieving successful implementation and maximum QoR. If a design is characterized by properly implemented hierarchy, it is much easier to decide which modules should comprise a partition.

Some common rules for creating good hierarchy for partitions include:

- Keep logic that needs to be optimized, implemented, and verified together in the same hierarchy. All designs have a hierarchy, but to achieve maximal performance, the hierarchy should be defined in the context of the FPGA's layout. This might require adding additional levels of hierarchy so that, for example, two critical modules can be synthesized and implemented in the same partition.
- Keep logic that needs to be packed together in the same level of hierarchy. This includes registers that need to be packed in larger components like block RAM and DSP. This rule also applies to I/O logic that needs to be packed together.
- Register the inputs and outputs of modules. All timing paths should be kept internal to a partition. A path that crosses a partition boundary can become critical because there are no logic optimizations across boundaries. Registering I/O is also important, because nets that cross the boundary from a preserved partition to an implemented partition are routed, not preserved.
- Do not use constants as inputs to partitions. Constants are not propagated across partition boundaries. Using input constants to disable unused features of a design in a partition can cause utilization and QoR issues. Unused logic is optimized away in the flat flow, but not when partitions are used.
- Do not allow unused inputs or outputs in the partition. Because the partition boundary is hard, unused inputs or outputs are not optimized away, which can lead to packing errors or higher utilization. If it is not possible to remove unused inputs or outputs from a module, create a wrapper that has only the used inputs and outputs. Put the partition on this wrapper, and the unused inputs and outputs of the critical module can then be optimized away.

The number of partitions is also important. Too many partitions can affect performance, because there are no optimizations across partition boundaries. In one design, for example, there might be just one partition on one very critical portion of the design. Other designs might have five or ten partitions. While this is all highly design dependent and there is no "magic number," 100 partitions is probably too many. The best candidates for partitions are modules or cores that have limited or no changes and that present difficulties in meeting timing.

Synthesis Flows

Partitions require that each logical module or entity be synthesized separately to prevent a design change in one area from causing different synthesis results in another area. This is accomplished by using either an incremental synthesis approach or a bottom-up synthesis approach.

Most third-party synthesis tools offer an incremental synthesis approach where each RTL module or instance is marked as a partition. When synthesis runs, each partition is synthesized separately with hard hierarchal boundaries, preventing an HDL change in one module from affecting another module. These tools also determine which modules or instances should be resynthesized based on HDL and/or constraint changes.

In a bottom-up synthesis flow, each module that will become a partition has a separate synthesis project and netlist. The user decides which synthesis projects need to be run based on their HDL code or synthesis constraint changes. The top level is synthesized by using black boxes for the lower level modules, and the lower level modules are synthesized without inferring I/Os or clocks. This approach is supported by third-party synthesis tools and XST.

Implementation Flows

The design preservation flow is supported in the PlanAhead™ software GUI and command line (batch files). In the ISE® software 12.1 and later, partitions are *not* supported in Project Navigator.

The steps for the PlanAhead software flow are:

1. Create a new project using "Specify synthesized (EDIF or NGC) netlist."
2. Define partitions in the Netlist tab of the Netlist Planner.
3. Run the Xilinx Implementation tools.
4. Promote partition results in the Designs Run window.
5. Promoted partitions are automatically set to Import for the next run.
6. Run the next iteration, setting the partition state to Implement for any partitions with a netlist change.

The steps for the command line flow are:

1. Create `xpartition.pxml` (can be copied from a PlanAhead software project).
2. Run implementation tools with `.pxml` file in the working directory.
3. Copy implementation results and `.pxml` file to an import directory.
4. Update partition state to Import in `.pxml` file and set the ImportLocation to the import directory.
5. Run the next iteration, setting the partition state to Implement for any partitions with a netlist change.

For more information on how to use the design preservation flow, see UG748, *Hierarchical Design Methodology Guide*.

Summary

Design preservation can reduce the implementation iterations during the timing closure phase by implementing just the changed modules. This is critical during the timing closure phase and at the end of the design cycle when small changes are being made. It can also reduce the verification time. To take advantage of these benefits, it is very important to think about the design's hierarchy during the RTL design phase.

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
03/31/10	1.0	Initial Xilinx release.

Notice of Disclaimer

The information disclosed to you hereunder (the "Information") is provided "AS-IS" with no warranty of any kind, express or implied. Xilinx does not assume any liability arising from your use of the Information. You are responsible for obtaining any rights you may require for your use of this Information. Xilinx reserves the right to make changes, at any time, to the Information without notice and at its sole discretion. Xilinx assumes no obligation to correct any errors contained in the Information or to advise you of any corrections or updates. Xilinx expressly disclaims any liability in connection with technical support or assistance that may be provided to you in connection with the Information. XILINX MAKES NO OTHER WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, REGARDING THE INFORMATION, INCLUDING ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT OF THIRD-PARTY RIGHTS.