Vivado Design Suite Tutorial:

Hierarchical Design

UG946 (v2013.4) December 18, 2013

Revision History

The following table shows the revision history for this document.

Date	Version	Changes
10/30/2013	2013.3	Initial Release
12/18/2013	2013.4	Manual was rewritten to highlight differences in the Hierarchical Design flow for the 2013.4 Vivado release.

Table of Contents

Revision History 2

Hierarchical Design Tutorial 4

Overview 4

Tutorial Design Description 4

Software Requirements 4

Hardware Requirements 4

Locating Tutorial Design Files 5

Lab: Top-Down Module Reuse 6

Introduction 6

Step 1: Define the Top-Level Module and Implementation 6

Step 2: Synthesize the Top-Level Module 8

Step 3: Define the OOC Modules and Implementations 9

Step 4: Synthesize the OOC Modules 10

Step 5: Defining the Top-Level Constraints 11

Step 6: Run the Top-Down Implementation 14

Step 7: Examine the Generated OOC Constraints 16

Step 8: Run the OOC and Top-Level Implementations 19

Step 9: Examine Other Tcl Scripts 20

Step 10: Verifying Timing Results 21

Conclusion 25

Overview

This Vivado® Hierarchical Design (HD) flow is only supported in the non-project batch flow. However, this tutorial will still use the Vivado IDE to create the required floorplan, timing, and context constraints. This methodology and these tools will help designers set up a design for Team Design parallel processing.

Tutorial Design Description

The small sample design used in this tutorial has a set of RTL design sources consisting of Verilog and VHDL. The VHDL sources are from multiple VHDL libraries. The design used throughout this tutorial contains:

- A RISC processor
- A pseudo FFT
- Gigabit transceivers
- Two USB port modules
- An xc7k70t device

Software Requirements

This tutorial requires that the 2013.4 Vivado Design Suite software release or later is installed. For installation instructions and information, see the *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (**UG973**).

Hardware Requirements

The supported Operating Systems include Redhat 5.6 Linux 64 and 32 bit, and Windows 7, 64 and 32 bit.

Xilinx® recommends a minimum of 2 GB of RAM when using the Vivado software on larger devices. For this tutorial, a smaller xc7k70t design is used, and the number of designs open at one time is limited. Although 1 GB is sufficient, it can affect performance.

Locating Tutorial Design Files

1. Download the `ug946-vivado-hierarchical-design-tutorial.zip` file from the Xilinx website:

   https://secure.xilinx.com/webreg/clickthrough.do?cid=351940&license=RefDesLicense&filename=ug946-vivado-hierarchical-design-tutorial.zip

2. Extract the zip file contents into any write-accessible location.

   The unzipped `Vivado_Tutorial_TD` data directory is referred to in this tutorial as `<Extract_Dir>`.

TIP: *The tutorial sample design data is modified while performing this tutorial. A new copy of the original `Vivado_Tutorial_TD` data should be used each time you start the tutorial.*

Introduction

This lab covers the Top-Down Module Reuse flow. This hierarchical design (HD) flow takes advantage of the top-level design to create the necessary constraints to drive the out-of-context (OOC) implementations. Modules identified as good candidates for reuse are implemented using the out-of-context (OOC) flow, but the necessary constraints are generated from an initial version of the full design. After the OOC implementations are complete, the results are read into the top-level implementation, preserving the placement and routing, to assemble the full design. This top-down constraint creation allows for timing driven creation of the necessary OOC context constraints, greatly increasing the quality of the OOC implementation results.

In this lab, you will incrementally define and run various stages of the flow. The definition and control of each stage is managed through a master Tcl file called `design.tcl`. In this Tcl script you will define the following:

- Modules to be synthesized using a bottom-up synthesis flow
- Top-Down implementation run to generate the OOC constraints
- OOC implementation runs for each partitioned module
- Top-level Reuse (assembly) run to import the OOC implementation results

Along the way, you will also use the Vivado IDE to define the necessary floorplan constraints.

Step 1: Define the Top-Level Module and Implementation

The Top-Down Module Reuse flow requires bottom-up synthesis results to prevent optimization between the top-level and partitioned modules. These bottom-up synthesis results are used to run the various implementation runs required for this flow.

Edit design.tcl

The `design.tcl` file will be used throughout this lab to define and control the synthesis and implementation of this design using the Top-Down Module Reuse flow. A completed version of this file, `design_complete.tcl`, is also provided, and can be used as a reference if necessary.

The intent of these scripts is to make `design.tcl` the only file you need to edit when setting up this flow for any design.

In `design.tcl` you can see the following types of information defined.

- Device/Package/Speed (xc7k70tfbg676-2)
- Flow Controls (synthesis, TopDown implementation, OOC implementation, HD/Reuse implementation, flat implementation)
- Input directories (RTL, PRJ, XDC, netlist, IP cores)
- Output directories (synthesis, implementation, checkpoints)
- Top Module definition (HDL files, XDC files, IP core files)
- OOC Module definition (HDL files, XDC files, IP core files)
- Top-level reuse/assembly implementation (import OOC implementations and implement the rest)
- OOC implementations (define OOC source, constraints, and implementation options)
- TopDown implementation (in-context implementation for generating OOC constraints)
- Flat implementation (to run a design without HD/OOC implementations)

To edit `design.tcl` for this tutorial design:

1. Open the file `<Extract_Dir>/design.tcl` in a text editor.
2. Note the variables defined in the top sections: Tcl Variables, Part Variables, and Setup Variables. These are variables that will not require any changes unless the directory structure changes, the desired target part changes, or you want to run only part of the flow (OOC synthesis or implementation, Top implementation, etc.).

The next section in `design.tcl`, Top Definition, is missing some information that needs to be defined. Only a few of the possible attributes (top_level, prj, vlog_headers) are needed for this design. Other designs may require fewer, or may require more attributes to be modified. For more information on the available module attributes, refer to the `README.txt` file in the `<Extract_Dir>/Tcl` directory.

TIP: *Any HDL source files such as VHDL or Verilog files can be defined in a list, However, this can become a bit messy. Alternatively, HDL source files can be defined in an input file of type PRJ. This PRJ is loosely based on the project file from the Xilinx ISE Design Suite XST synthesis tool, and can easily be created in a text editor. This tutorial uses PRJ files for all modules. Defining the PRJ variable will override any Verilog or VHDL files listed, so the PRJ must contain the comprehensive list of all files to be synthesized by Vivado synthesis.*

3. Define the following values for the listed module and implementation attributes for top. Note some of these variables are already assigned for you. The lines that need to be added/modified are bold. Tcl is case sensitive, so be aware of this when completing the missing fields in `design.tcl`. Also, a completed file called `design_complete.tcl` is provided with the tutorial, and can be used as a reference.

```
set top "top"
add_module $top
set_module_attribute $top    top_level     1
set_module_attribute $top    prj           $prjDir/$top.prj
set_module_attribute $top    vlog_headers  [list \
                                          $rtlDir/or1200/or1200_defines.v \
                                          $rtlDir/usbf/usbf_defines.v     ]
set_module_attribute $top    synth          ${run.topSynth}

add_implementation $top
set_impl_attribute $top      top           $top
set_impl_attribute $top      implXDC       [list $xdcDir/${top}.xdc]
set_impl_attribute $top      impl          ${run.topImpl}
set_impl_attribute $top      hd.impl       1
```

IMPORTANT: *Unused variables are defined with a default value. Only define the module/implementation variables that are necessary for the current design.*

As mentioned previously, the HDL required to synthesize the top-level module in this design are listed in the PRJ file defined by the `prj` module attribute. The control to run synthesis on the module is controlled by the `synth` attribute, and assigned to one of the flow control variables at the top of `design.tcl` (for easier access).

For the top-level implementation, we defined the top-level module name and an XDC file to be used for implementation. The attributed that controls whether a defined implementation runs or not is the `impl` attribute, and the `hd.impl` attribute tells the scripts that this implementation will be importing OOC implementation results.

4. Save your changes to `design.tcl`.

Step 2: Synthesize the Top-Level Module

Now that the top-level module has been defined, we can synthesize it. The synthesis results of top will have black box modules for the two OOC modules (usbf_top and or1200_top).

1. To synthesize the top-level module, you now need to set the flow control variable to 1. This variable is defined near the top of `design.tcl`, in the Flow Control section, and is called "run.topSynth". Change the value of this variable to be 1, and save `design.tcl`. The Flow Control section should look like the following.

```
####flow control
set run.topSynth   1
set run.oocSynth   0
set run.tdImpl     0
set run.oocImpl    0
set run.topImpl    0
set run.flatImpl   0
```

2. From a shell/prompt, source `design.tcl` in batch mode using the following command.

```
vivado -mode batch -source design.tcl -notrace
```

IMPORTANT: *The -notrace option is not required, but will prevent the contexts of the provided Tcl scripts from being echoed to the console. Using -notrace makes it easier to see what the scripts are actually doing by keeping the console cleaner.*

If everything was correctly defined in `design.tcl`, the synthesis of top will complete, and the results will be saved to `./Synth/top/top_synth.dcp`. If something was not correctly defined, you may get errors from either the scripts or Vivado. In this case read the error messages carefully, and reference `design_complete.tcl` if necessary.

Step 3: Define the OOC Modules and Implementations

Each OOC module will be synthesized bottom-up, and then each instance of the OOC module will have its unique OOC implementation. In the following steps you will define and examine the necessary OOC module and OOC implementation definitions for this design. These modules are processed as summarized below.

- **usbf_top**
  OOC module to be synthesized bottom-up
- **usbEngine0**
  OOC implementation of usbf_top with unique Pblock range
- **usbEngine1**
  OOC implementation of usbf_top with unique Pblock range
- **or1200_top**
  OOC module to be synthesized bottom-up
- **cpuEngine**
  OOC implementation of cpuEngine with unique Pblock range

1. Define the necessary module and OOC implementation attributes for `usbf_top`. Again, some of these attributes have already been defined for you, and the lines you need to complete are bold.

```
set module1 "usbf_top"
add_module $module1
set_module_attribute $module1 prj          $prjDir/$module1.prj
set_module_attribute $module1 synth        ${run.oocSynth}

set instance "usbEngine0"
add_ooc_implementation $instance
set_ooc_attribute $instance   module       $module1
set_ooc_attribute $instance   inst         $instance
set_ooc_attribute $instance   hierInst     $instance
set_ooc_attribute $instance   implXDC      [list \
     $xdcDir/${instance}_phys.xdc         \
     $xdcDir/${instance}_ooc_timing.xdc   \
     $xdcDir/${instance}_ooc_budget.xdc   \
     $xdcDir/${instance}_ooc_optimize.xdc \
]
set_ooc_attribute $instance   impl         ${run.oocImpl}
set_ooc_attribute $instance   preservation routing
```

Note that the second OOC implementation of `usbf_top` (`usbEngine1`) is defined identically to the first implementation of `usbf_top` (`usbEngine0`), except for the value of variable `$instance`.

2. Examine the definition of OOC module `or1200_top`, and its OOC implementation `cpuEngine.` This section is already complete, and there are no modification required for this OOC module or its implementation.

Step 4: Synthesize the OOC Modules

So far you have defined the top-level module and synthesized it, and you have defined the OOC modules. You will now modify `design.tcl` to run the OOC module synthesis runs, which will result in a fully synthesized design.

1. To synthesize the OOC modules, you now need to set the flow control variable to 1. This variable is defined near the top of `design.tcl`, in the Flow Control section, and is called "run.oocSynth".

   Change the value of this variable to 1, and change the value of the variable "run.topSynth" back to 0. Failure to change the top-level synthesis variables back to 0 will result in top-level synthesis being run again. The Flow control section of `design.tcl` should look like the following.

   ```
   ####flow control
   set run.topSynth   0
   set run.oocSynth   1
   set run.tdImpl     0
   set run.oocImpl    0
   set run.topImpl    0
   set run.flatImpl   0
   ```

2. Save the changes to `design.tcl`.

3. From a shell/prompt, source `design.tcl` in batch mode using the following command.

   ```
   vivado -mode batch -source design.tcl -notrace
   ```

If everything was correctly defined in `design.tcl`, the synthesis of top will complete, and the results will be saved to `./Synth/<module>/<module>_synth.dcp`. If something was not correctly defined, you may get errors from either the scripts or Vivado. In this case read the error messages carefully, and reference `design_complete.tcl` if necessary.

Step 5: Defining the Top-Level Constraints

This Tutorial uses a Top-Down approach to generate the necessary constraints for the OOC implementations. If the top-level module and constraints are available, you can take advantage of this to automatically generate a set of complex OOC implementation constraints. The full in-context design will be used to generate the constraints for the OOC implementations. The top-level constraints have already been created for this tutorial design, including a pinout, Pblocks, and clock timing and location constraints. In this section, you will load the full in-context design and examine the existing top-level constraints.

Load the Top-Level Design and Constraints

1. Open the top-level synthesis results in the Vivado IDE by running the following command from a shell/prompt.

   ```
   vivado ./Synth/top/top_synth.dcp
   ```

   Note that when the design loads, the three instances of the OOC modules are shown as black boxes in the Netlist window. You have only loaded the top-level synthesis result, so this is expected. You could fill in these black boxes using the command `read_checkpoint -cell` and referencing the module synthesis results, but it is not necessary for this step.

2. Click on the Device view tab, and note that there are no Pblocks in the design.

3. Load in the top-level XDC file with timing, I/O, and Pblock constraints by typing this command at the Tcl Console.

   ```
   read_xdc ./Sources/xdc/top_flpn.xdc
   ```

You should now see three Pblocks defined in the Device view. If the design did not already have Pblock and/or I/O pin constraints, you would use the Vivado IDE (the GUI) with the design loaded to create this XDC. For more information on using the Vivado IDE to create Pblock constraints, refer to the *Vivado Design Suite User Guide: Getting Started* (**UG910**).

When the floorplan is complete, it can be exported using the `write_xdc` command from the Tcl Console, or the **File >Export >Export Constraints** option from the Vivado IDE's Menu bar. Since the floorplan is already defined for this tutorial design, this is not necessary.

Examine the Pblock Constraints

In this design, there are two OOC modules, representing three instances, which require Pblock constraints:

- usbEngine0 (usbf_top)
- usbEngine1 (usbf_top)
- cpuEngine (or1200_top)

Based on the pinout of this tutorial design, these Pblocks have already been created. However, the following steps will have you examine the Pblock constraints, and some additional constraints that aid in the OOC implementation.

1. Open the top-level XDC file, `./Sources/xdc/top_flpn.xdc`, in a text editor.
2. Search the XDC file for the three Pblocks. Locate the Pblock constraints and note the syntax.
3. Note the additional CONTAIN_ROUTING constraints on each of the OOC module Pblocks. The use of this constraint is highly recommended, and should be set on all Pblocks associated with an OOC module. This allows the tools to control the routing in the OOC implementation (just as the Pblock RANGE controls placement) so that no routing conflicts occur when the final design is assembled. The CONTAIN_ROUTING constraint is specific to a Pblock and must come after the `create_pblock` commands in the XDC file.

```
set_property CONTAIN_ROUTING true [get_pblocks pblock_usbEngine0]
set_property CONTAIN_ROUTING true [get_pblocks pblock_usbEngine1]
set_property CONTAIN_ROUTING true [get_pblocks pblock_cpuEngine]
```

Examine the HD.PARTPIN_RANGE Constraints

All ports of an OOC module (except clock ports or ports connected to dedicated logic such as I/O buffers) will have a partition pin (PartPin) to help guide the placement and routing of the module. If the scripts do not find any HD.PARTPIN_RANGE constraints on the module pins, it will create one assigned to match the slice range of the OOC module's Pblock.

In order to get higher quality results from the OOC implementations, the placement of these ports can be guided by providing more specific HD.PARTPIN_RANGE constraints. In this tutorial, the PartPins are ranged to particular edges of the Pblocks using multiple SLICE range values. In a later step, you will run `place_design` to get the in-context placement of these PartPins.

1. Examine `top_flpn.xdc` and note the following HD.PARTPIN_RANGE constraints.

-
  ```
  set_property HD.PARTPIN_RANGE {SLICE_X0Y97:SLICE_X23Y99
  SLICE_X21Y0:SLICE_X23Y99} [get_pins usbEngine0/*]
  ```

-
  ```
  set_property HD.PARTPIN_RANGE {SLICE_X0Y100:SLICE_X23Y102
  SLICE_X21Y100:SLICE_X23Y199} [get_pins usbEngine1/*]
  ```

-
  ```
  set_property HD.PARTPIN_RANGE {SLICE_X36Y97:SLICE_X61Y99
  SLICE_X59Y0:SLICE_X61Y99} [get_pins cpuEngine/*]
  ```

Embedded I/O Constraints

This design has direct connections from top-level ports to the OOC modules, and instantiates I/O buffers inside of the OOC modules. Whenever a direct connection (fanout of 1 for inputs) exists between an OOC module and a top-level port, it is recommended that the I/O buffers are embedded in the OOC module. This will provide better results since the tools have more information about how the OOC module is connected, and can place input and output logic in the dedicated IOLOGIC blocks.

Note that this does require top-level synthesis to prevent buffers from being inferred on ports with embedded I/O. You can see this in this tutorial design by searching the top-level source file, `./Sources/hdl/top.v`, for IO_BUFFER_TYPE attributes. For more information on how to control buffer insertion, please refer to the *Vivado Design Suite User Guide: Synthesis* (**UG901**).

Examine the embedded I/O and XDC constraints.

1. If it is not open already, open the XDC file, `top_flpn.xdc`, and note these PACKAGE_PIN and IOSTANDARD constraints:

```
set_property IOSTANDARD LVCMOS18 [get_ports {DataIn_pad_0_i[0]}]
set_property PACKAGE_PIN G24 [get_ports {DataIn_pad_0_i[0]}]
```

Some of the ports to which these constraints are applied connect directly to OOC module pins. As you can see in the design schematic below, the **DataIN_Pad_i[7:0]** input bus is connected directly to the **usbEngine0** instance, which contains the input buffers. This will allow for the OOC implementation to make the best decisions it can around placing related logic at or near the I/O sites associated with these PACKAGE_PIN constraints.

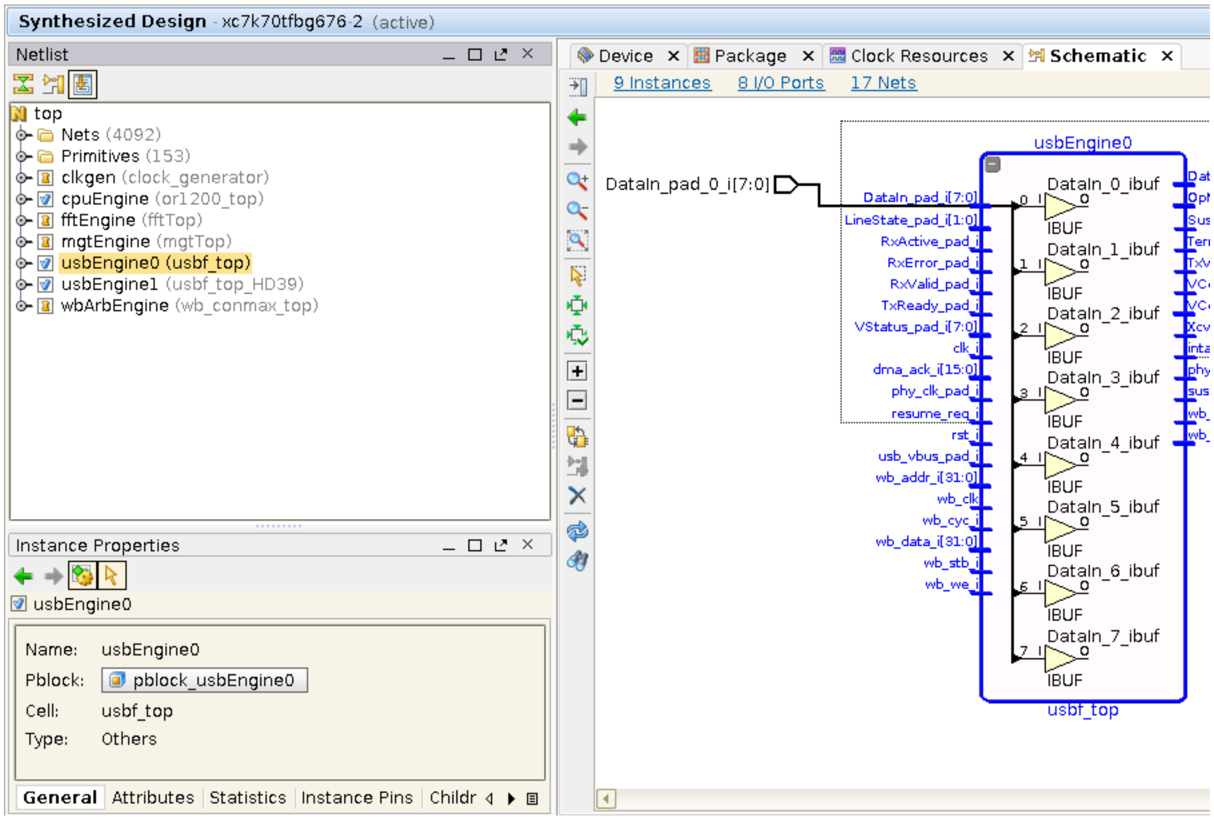


Figure 1: Schematic View

Since these I/O buffers are part of the OOC module, they need to be properly constrained in the OOC implementation. Because these module pins connect directly to the top-level ports, the constraints on the top-level ports can be propagated to the module pins. These physical constraints will be set later in this tutorial and will be examined in the **Examine the Physical XDC Constraints** section.

Step 6: Run the Top-Down Implementation

Using the in-context design and the provided commands and scripts, all of the necessary constraints to run the OOC implementations can be created automatically. The implementation in which these OOC constraints are generated is referred to as the Top-Down implementation. In this section you will set up and run this implementation by sourcing `design.tcl` with the correct flow controls set. This will generate the following XDC files for each instance of the OOC modules.

- `<instance>_phys.xdc`
  Physical XDC constraints including Pblocks, I/O, lock, HD.PARTPIN_LOCS
- `<instance>_ooc_optimize.xdc`
  Defines `set_logic_*` optimization constraints to tie off constant inputs, or unconnected outputs.
- `<instance>_ooc_timing.xdc`
  Clock, clock source, clock uncertainty, asynchronous clock groups (if applicable), and clock latency constraints.
- `<instance>_ooc_budget.xdc`
  Defines set_max_delay constraints to and from the OOC module ports to the interface logic inside of the OOC module. The requirement of these constraints is set to 50% of the clock period by default. This value can be adjusted.

Note the optimize, timing, and budget XDC files have the string "`ooc`" in their names. This is so the scripts can detect these and mark them for OOC use only. Without this designation, the OOC timing constraints will be imported into the top-level design, potentially causing constraint interaction issue and incorrect timing reports. For more information on filtering out OOC specific constraints, please refer to the USED_IN property in the *Vivado Design Suite User Guide: Hierarchical Design* (**UG905**).

1. Open `design.tcl` and view the TopDown implementation section (or study the text below).

```
####################################################################
### Create TopDown implementation run
####################################################################
set module1File "$synthDir/$module1/${module1}_synth.dcp"
set module2File "$synthDir/$module2/${module2}_synth.dcp"
add_implementation TopDown
set_impl_attribute TopDown      top          $top
set_impl_attribute TopDown      implXDC      [list $xdcDir/${top}_flpn.xdc]
set_impl_attribute TopDown      td.impl      1
set_impl_attribute TopDown      cores        [list $module1File             \
                                     $module2File              \
                                     [get_module_attribute $top cores]     \
                                     [get_module_attribute $module1 cores] \
                                     [get_module_attribute $module2 cores] \
                                             ]
set_impl_attribute TopDown      impl         ${run.tdImpl}
set_impl_attribute TopDown      route        0
```

Note that the synthesized results for the two OOC modules are added as cores for this run, and that the XDC files provided are the floorplanned versions of the top-level with Pblocks, CONTAIN_ROUTING, and HD.PARTPTIN_RANGE constraints.

Also note that the implementation attribute `impl` is controlled by flow control variable `$run.tdImpl`, and that the attribute `route` is set to '0'. Since the point of this top-down run is to place the PartPins and to generate the OOC constraints, it is not necessary to run `route_design` for this implementation.

2. Edit `design.tcl` to run the top-down implementation. Do this by making the flow control section look like the following.

```
####flow control
set run.topSynth   0
set run.oocSynth   0
set run.tdImpl     1
set run.oocImpl    0
set run.topImpl    0
set run.flatImpl   0
```

3. From a shell/prompt, source `design.tcl` in batch mode using this command:

```
vivado -mode batch -source design.tcl -notrace
```

Running the above command will run the top-down implementation. Below is a list of commands done for each OOC module instance during this implementation.

- Set the property HD.PARTITION to define hierarchical boundaries.
- Call Tcl proc `create_set_logic` to create boundary optimization constraints.
- Call Tcl proc `create_ooc_clocks` to generate timing constraints file.
- Run in-context implementation to get placement of PartPins.
- Call Tcl proc `write_hd_xdc` to write out the interface timing and budget constraints, and to export PartPin locations and other physical constraints.

The above Tcl procs can be found in the provided Tcl scripts at:

```
<Extract_Dir>/Tcl/hd_floorplan_utils.tcl
```

These Tcl procs are called automatically by the Top-Down implementation which uses the `impl` Tcl proc defined in `<Extract_Dir>/Tcl/impl.tcl`.

Step 7: Examine the Generated OOC Constraints

The following subsections provide details about the constraints that were created in the Top-Down implementation.

Examine the Timing XDC Constraints

The following constraints may be added to the timing XDC file by the `create_ooc_clocks` Tcl proc.

- create_clock
- HD.CLK_SRC
- set_system_jitter
- set_clock_uncertainty
- set_clock_latency
- set_clock_groups

The create_clock constraints are required to define the clocks on local clock ports. HD.CLK_SRC is a required context constraint (for clocks whose driver is outside of the OOC module) in order to calculate clock pessimism removal. It is highly recommended to lock down all clocking logic in both the top and OOC modules, and to provide HD.CLK_SRC constraints for any clocks driven by clock buffers in the top-level design.

In this design all OOC module clocks are driven by global buffers in the top level. That means that every clock port on the OOC modules should have an HD.CLK_SRC constraint. This design does already have the BUFG locations set in the top-level XDC, and this is a requirement to get the `create_ooc_clocks` Tcl proc to generate these constraints.

It is also required to correctly define clock uncertainty values on each module clock, as well as clock uncertainty values between synchronous clocks. This again can be derived from the in-context design, and these constraints will be written to the timing XDC file. Because this user-defined uncertainty value already includes the system jitter, the system jitter value should be set to zero for the OOC module.

To get accurate skew estimates on clocks that are driven from outside of the OOC module, clock latency constraints must be defined. The constraints written out to the timing XDC file may need to be adjusted for some cases. Currently these values are hard coded to 100ps difference between min and max values.

Finally, if clocks are defined as asynchronous in the top-level design, or if multiple clocks exist for a single module port (driven by a BUFGCTRL), then clock group constraints will be generated by the `create_ooc_clocks` Tcl proc. However, this particular design does not have these cases.

1. In your text editor, open the timing XDC file generated by the top-down implementation for `usbEngine0`. This file will be located here:

   ```
   <Extract_Dir>/Sources/xdc/usbEngine0_ooc_timing.xdc
   ```

Examine the Physical XDC Constraints

The following constraints may be added to the physical XDC file by the `write_xdc -cell` command.

- Pblock
- I/O constraints (IOSTANDARD, PACKAGE_PIN, SLEW, IOB)
- set_logic_zero, set_logic_one, and set_logic_unconnected
- LOC constraints (lock values for BRAM or other floorplanned logic)
- HD.PARTPIN_LOCS (for module ports with PartPins)

The Pblock defined on the specified cell will be written out to the physical XDC file. This includes all Pblock properties including the ranges, CONTAIN_ROUTING, and `add_cells_to_pblock`. If the Pblock is correctly defined on the hierarchical instance in the top level, then the `add_cells_to_pblock` in the cell XDC will use the `-top` option. This tells the tools that the current top-level (and everything under the current hierarchy) belongs to the Pblock.

As seen earlier in the tutorial, this design does have embedded I/O buffers in the OOC modules, and therefore any constraints on these embedded I/O are also added to the physical XDC.

HD.PARTPIN_LOCS is a key constraint to control the OOC implementation. Using the in-context design we can run an in-context implementation to generate the HD.PARTPIN_LOCS. Specifically, we need to run `place_design` on the in-context design. Before running the `write_xdc -cell` command, the scripts run `opt_design` and `place_design` on the in-context design to get the PartPin placement.

1. Open the physical XDC file generated by the top-down implementation for cpuEngine. This file will be located at `<Extract_Dir>/Sources/xdc/cpuEngine_phys.xdc`.

Examine the Optimization XDC Constraints

The following constraints may be added to the optimize XDC file by the `create_set_logic` Tcl proc:

```
set_logic_zero/set_logic_one/set_logic_unconnected
```

The `set_logic_*` constraints are context constraints to aid in optimization of the OOC module. These constraints tell the implementation tools which input ports are tied to constants (power or ground), or which output ports are left unconnected. Without these constraints, optimization across the OOC boundary cannot occur, and the quality of the OOC implementation results may be impaired. For this tutorial design, only the instance cpuEngine has ports that need these optimization constraints.

1. Open the optimization XDC file generated by the top-down implementation for cpuEngine. This file will be located at `<Extract_Dir>/Sources/xdc/cpuEngine_ooc_optimize.xdc`.

Overview of Interface Budget XDC Constraints

The interface budget XDC file is generated until the OOC implementations are run. The budget XDC contains max_delay constraints for all module ports that have a PartPin. The HD.PARTPIN_LOCS constraints are an important part of controlling the placement of the OOC module, but interface timing constraints are also required to control how closely the interface logic gets placed to the PartPin locations. Without the interface timing constraints there is no guarantee that module interface logic will be placed close to the associated PartPin.

1. Examine the interface budget XDC file generated during **Step 4: Synthesize the OOC Modules**. This file will be located here:

   `<Extract_Dir>/Sources/xdc/usbEngine1_ooc_budget.xdc`.

   The default value of these set_max_delay constraints is 50% of the period. This can be adjusted with the `-percent` option of `::debug::gen_hd_timing_constraints`, which is called from `<Extract_Dir>/Tcl/ooc_impl.tcl`. These budget constraints are intended as a template, and may need to be adjusted on a per port and per bus basis to meet design requirements.

Examine the PartPin Locations

The PartPins will be placed during `place_design` by the timing driven, in-context run. The placement of the PartPins is controlled by the HD.PARTPIN_RANGE specified in the top-level XDC.

To see the location of the PartPins picked by the in-context, top-down implementation, load the placed DCP into the Vivado IDE and examine the locations visually.

1. Load the placed DCP from the top-down implementation by typing the following command from a shell/prompt.

   ```
   vivado ./Implement/TopDown/top_place_design.dcp
   ```

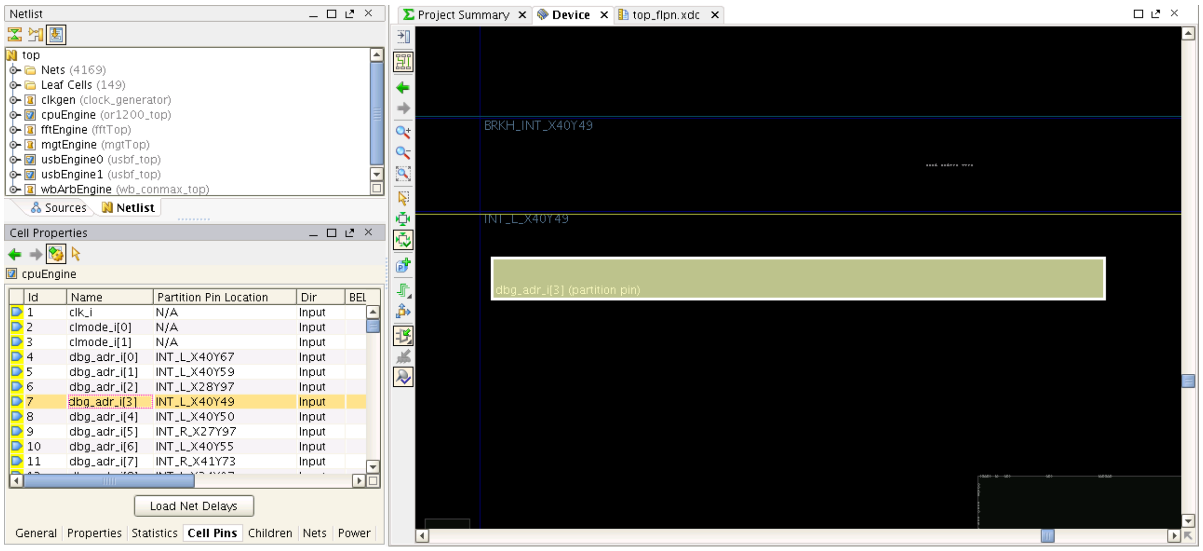
2. Select an instance of an OOC module, such as usbEngine0, in the Netlist window.
3. In the Properties window select the **Cell Pins** tab.
4. Select a pin, and note the Partition Pin Location field (some pins such as clocks or ports with embedded I/O will not have PartPins, and the value will be N/A). Selecting a pin with a PartPin location should also cause the device view to zoom to the selected location. If the device view does not automatic zoom to the selected Partition Pin, verify that the Auto Fit Selection feature is turned on by clicking the Auto Fit Selection icon () in the Device window.
5. Modify the value by doing one of the following:
   - Drag and Drop the pin from the Cell Properties window to the Device View.
   - Drag and Drop the pin from the current location in the Device view to a new location in the Device view.
   - From the Tcl Console, modify the property using the `set_property` command:

```
set_property HD.PARTPIN_LOCS INT_L_X40Y73 [get_pins{cpuEngine/dbg_adr_i[7]}]
```

   - Edit the `<instance>_phys.xdc` file (created by the top-down implementation) in a text editor.

   If desired, the modified PartPin location constraints could be exported using the `write_xdc -cell` command to overwrite the previous `<cell>_flpn.xdc` file created during the top-down implementation.

6. Close the Vivado IDE and `top_place_design.dcp`. Click on the **Don't Save** option when exiting, since all information is stored in the OOC XDC files.



Id	Name	Partition Pin Location	Dir	BEL
1	clk_i	N/A	Input	
2	clmode_i[0]	N/A	Input	
3	clmode_i[1]	N/A	Input	
4	dbg_adr_i[0]	INT_L_X40Y67	Input	
5	dbg_adr_i[1]	INT_L_X40Y59	Input	
6	dbg_adr_i[2]	INT_L_X28Y97	Input	
7	dbg_adr_i[3]	INT_L_X40Y49	Input	
8	dbg_adr_i[4]	INT_L_X40Y50	Input	
9	dbg_adr_i[5]	INT_R_X27Y97	Input	
10	dbg_adr_i[6]	INT_L_X40Y55	Input	
11	dbg_adr_i[7]	INT_R_X41Y73	Input	

Figure 2: PartPin Locations

Step 8: Run the OOC and Top-Level Implementations

In this section you will modify `design.tcl` to run the OOC and top-level assembly runs. The OOC implementation results must exist before the top-level assembly can be run, but these can be all set to run at the same time. The scripts take care of the ordering. In fact, if you examine `design_complete.tcl` you will notice the flow control variables are all set to 1. If the necessary top-level constraints exist for the design, the entire flow can be run over by simply setting all of the flow control variables to 1, and then sourcing `design.tcl`.

1. Edit the flow control section in `design.tcl` to turn on the OOC and top-level implementations, and turn off everything else.

```
####flow control

set run.topSynth   0
set run.oocSynth   0
set run.tdImpl     0
set run.oocImpl    1
set run.topImpl    1
set run.flatImpl   0
```

2. Save the changes to `design.tcl`.

3. From a shell or prompt, source `design.tcl` in batch mode.

```
vivado -mode batch -source design.tcl –notrace
```

TIP: *In the above command,* `design.tcl` *can be replaced by* `design_complete.tcl` *to run a completed version of the lab.*

This step may take approximately half an hour to complete. While this is running, you can move on to the next step to examine the additional Tcl files provided with this tutorial.

Step 9: Examine Other Tcl Scripts

This flow is supported in a non-project (Tcl or batch) mode only. Scripts to run the flow are provided in the archive, and are intended to be used as a starting point for any design using this flow. You can always create your own scripts, or modify the provided scripts, but if you are new to Vivado and Tcl, the provided scripts are an easy entry point into this flow.

The provided scripts also provide a few helpful features including:

- `run.log`
  A log file containing runtime and timing information for each implementation
- `command.log`
  A log file containing all commands issued during the flows
- `critical.log`
  A log file containing all Critical Warnings found during the flow
- A robust and flexible environment, designed to minimize scripting errors, and focus on design issues and tool issues.
- A single file to define all design information and modify flow controls.

All information about the design exists in the Tcl file `design.tcl`. Ideally, this is the only Tcl file you need to edit in order to run this flow with any design. This section will provide you with information about the additional Tcl scripts provided with this tutorial in case you want to modify or change anything in the provided scripts.

Besides `design.tcl`, the rest of the Tcl files provided in the tutorial are generic to all designs and all HD flows (including Partial Reconfiguration). The files are not intended to be edited, and exist in the `<Extract_Dir>/Tcl` directory. Below is a list of the remaining Tcl files and a description of what they do.

- `README.txt`
  Gives a summary of the command and attributes supported by the scripts. Briefly describes the design purpose of module and implementation attributes as well as a quick description of the provided Tcl files.
- `design_utils.tcl`
  Defines all valid module and implementation settings, as well as procs to validate, set, and return these values.

- `run.tcl`
  This is the main script that drives the flow. There are implementation control variables in this file to control which parts of implementation to run for Top and each OOC module.
- `synth.tcl`
  Called by `run.tcl` if any module has the synth attribute set to 1. Calls `synth_design` with `-mode out_of_context` for any modules not defined as top_level.
- `synth_utils.tcl`
  Contains a variety of Tcl procs used to process XDC, HDL, and PRJ files for synthesis.
- `impl.tcl`
  Called by `run.tcl` for any implementation with the impl attribute set to 1. Used for top-down, assembly (Module Reuse), or flat implementation runs.
- `ooc_impl.tcl`
  Called by `run.tcl` for any OOC implementations with the impl attribute set to 1. Used for module implementation flows to implement a module OOC.
- `pr_impl.tcl`
  Not used for this tutorial; used to implement Partial Reconfiguration configurations. Called by `run.tcl` for any configuration with the impl attribute set to 1.
- `impl_utils.tcl`
  Contains a variety of Tcl procs used to implement various implementation flows supported by the scripts.
- `step.tcl`
  Defines a Tcl proc (`impl_step`) called by `impl.tcl` to call the various phases of implementation. Runs the specified implementation step and writes out reports and checkpoints. If no design is open in memory, this proc tries to open a checkpoint for the previous step. In this way you can break up implementation for debug and analysis.
- `log.tcl`
  Defines Tcl procs used to create a `run.log` file. This file contains information including a list of params that were set by the scripts, run time information for synthesis and implementation, and final timing numbers reported by `route_design`.

Step 10: Verifying Timing Results

Once the OOC and top-level implementations are done, the results can be examined using the Vivado IDE. Even though this lab uses the non-project batch flow, the IDE can still be used to debug or view results. The scripts provided with this tutorial write out checkpoints at various stages of the design, and any of those checkpoints can be loaded into the IDE for viewing.

In addition to this, the top-level and each OOC implementation generate a few key report files along the way, and write these files out to this directory:

```
<Extract_Dir>/Implementation/<module>/reports
```

Open a Checkpoint in the Vivado IDE

As mentioned previously, a checkpoint can be loaded into memory for debug or analysis. This can be done in Tcl or GUI mode.

To open a checkpoint in Tcl Mode:

1. From the directory `<Extract_Dir>`, type the following command:

   ```
   vivado -mode tcl ./Implement/top/top_route_design.dcp
   ```

2. Interact with the design using commands such as `report_utilization` or `report_route_status`.
3. Start the Vivado IDE by typing `start_gui`.

Note that at any time you can close the IDE and return to the Tcl prompt by typing `stop_gui` in the Tcl Console.

When the placed and routed design is loaded into the IDE, you can right-click on modules in the Netlist view and select **Highlight Primitives**. **Figure 3** is an image of the final routed top design with the USB and CPU instances highlighted. Note how each OOC module is contained within its quadrant but the top-level logic is interspersed throughout the device.

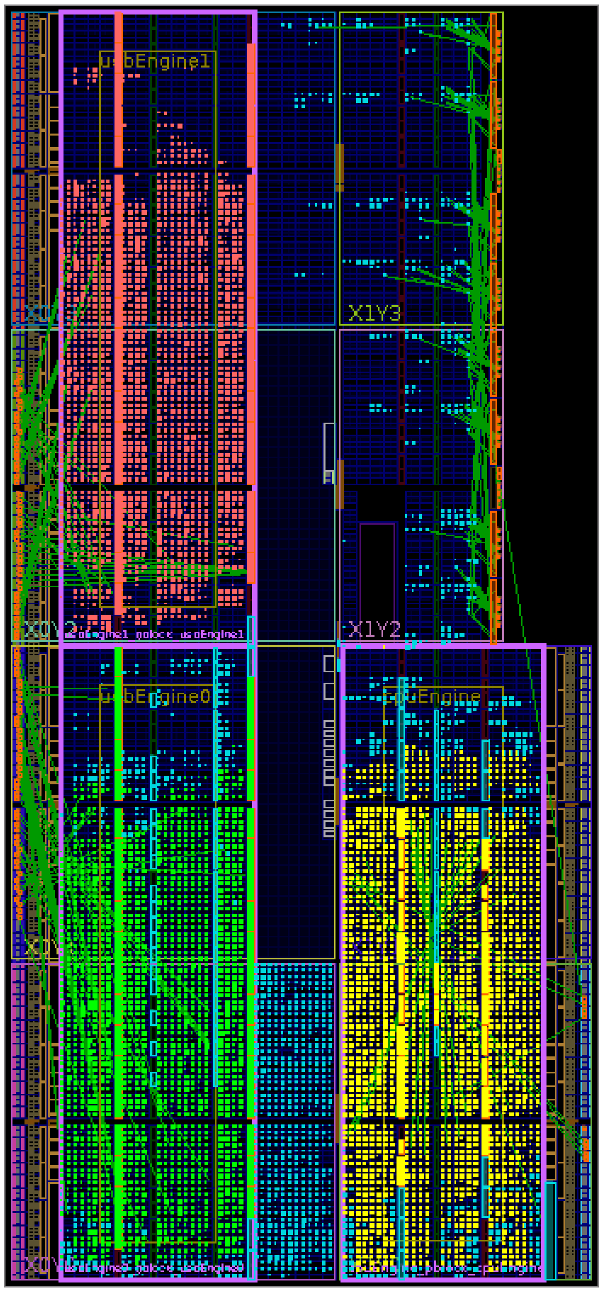


Figure 3: Final Assembled Design

Verify Timing Results

Timing results can be viewed using the IDE or from the reports generated by the Tcl scripts.

Verify timing by looking at the `report_timing_summary` report created by the scripts. Verify the Total Negative Slack (TNS) is `0.000`.

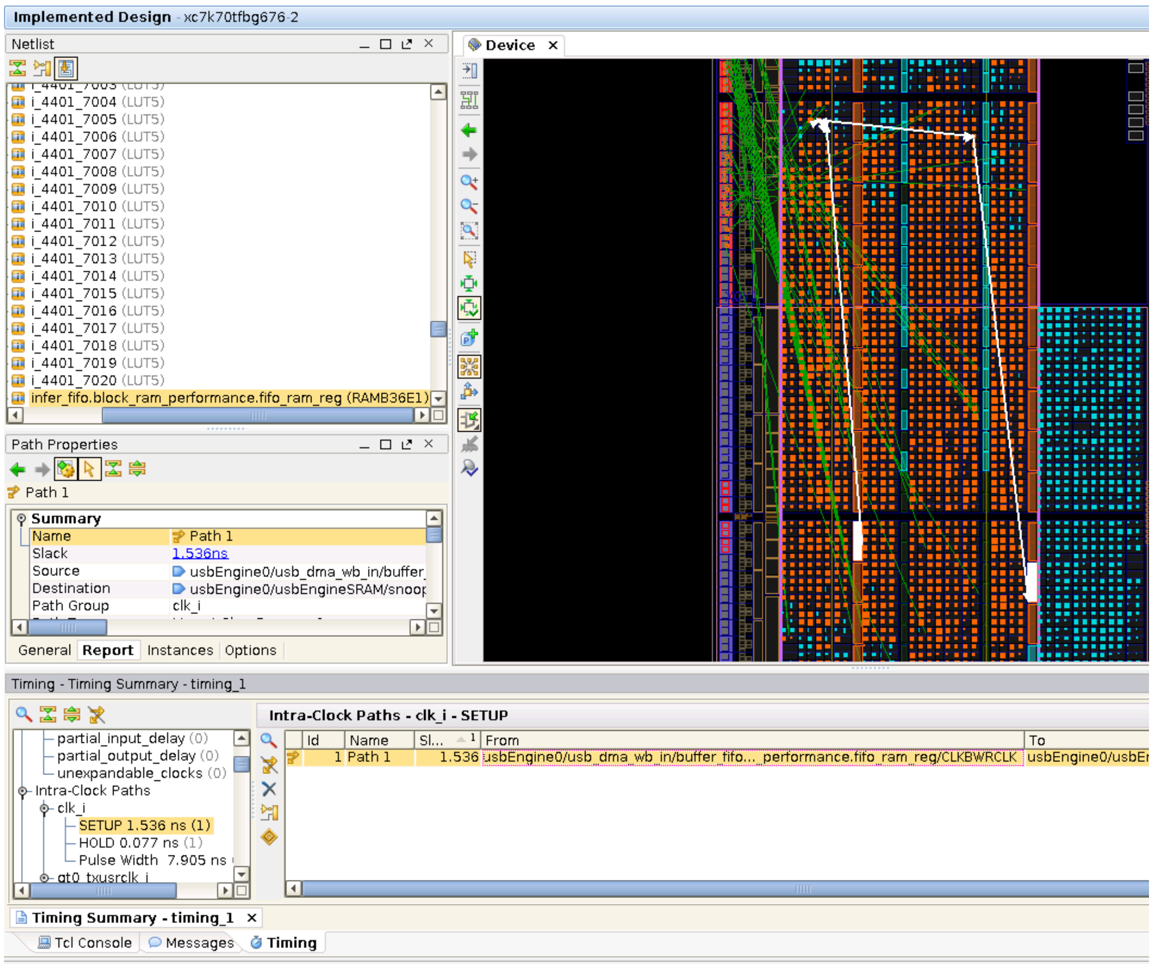
1. In a text editor, open the `report_timing_summary` report file:

```
<Extract_Dir>/Implementation/top/reports/top_timing_summary_route_design.rpt
```

2. Browse down to the Design Timing Summary and verify the TNS value is `0.000`.
3. Explore other sections of the timing summary report to familiarize yourself with the information available.
4. Verify timing interactively in the Vivado IDE by running `report_timing_summary` with a name option.
5. With the Vivado IDE open and a post-route `route_design` checkpoint loaded, run `report_timing_summary` from the Tcl Console:

```
report_timing_summary -name timing_1
```

Note that using the `-name` option will open a Timing Summary window with a summary, and paths that be selected to interactively view and debug timing problems.



Implemented Design - xc7k70tfbg676-2
Netlist
Device
Path Properties
Path 1
General
Report
Instances
Options
Timing - Timing Summary - timing_1
Timing Summary - timing_1
Tcl Console
Messages
Timing

i_4401_7004 (LUT5)
i_4401_7005 (LUT5)
i_4401_7006 (LUT5)
i_4401_7007 (LUT5)
i_4401_7008 (LUT5)
i_4401_7009 (LUT5)
i_4401_7010 (LUT5)
i_4401_7011 (LUT5)
i_4401_7012 (LUT5)
i_4401_7013 (LUT5)
i_4401_7014 (LUT5)
i_4401_7015 (LUT5)
i_4401_7016 (LUT5)
i_4401_7017 (LUT5)
i_4401_7018 (LUT5)
i_4401_7019 (LUT5)
i_4401_7020 (LUT5)
infer_fifo.block_ram_performance.fifo_ram_reg (RAMB36E1)

Summary

Name	Path 1
Slack	1.536ns
Source	usbEngine0/usb_dma_wb_in/buffer_
Destination	usbEngine0/usbEngineSRAM/snoop
Path Group	clk_i

partial_input_delay (0)
partial_output_delay (0)
unexpandable_clocks (0)
Intra-Clock Paths
clk_i
SETUP 1.536 ns (1)
HOLD 0.077 ns (1)
Pulse Width 7.905 ns
gt0_txusrclk_i

Intra-Clock Paths - clk_i - SETUP

Id	Name	Sl...	From	To
1	Path 1	1.536	usbEngine0/usb_dma_wb_in/buffer_fifo... performance.fifo_ram_reg/CLKBWRCLK	usbEngine0/usbEn

Figure 4: Interactive Timing Summary

Checking the Log Files

When running the provided scripts, log files are generated that are worth note.

1. In a text editor, open the `run.log` file. This file captures run time and timing information (if available) for each phase of the flow that ran the last time `design.tcl` was sourced.
2. In a text editor, open the `command.log` log file. Note the list of every command that was used to run the flow. This log file can be modified and sourced at the Vivado command line to test or rerun certain parts of the flow.

   ```
   <Extract_Dir>/command.log
   ```
3. In a text editor, open the `critical.log` log file. This file is a collection of all Critical Warnings found during the implementation flow. Some Critical Warnings are expected and acceptable, but all messages should be reviewed to make sure that no action is required to correct a serious problem.

   ```
   <Extract_Dir>/critical.log
   ```

Note that all of these log files will automatically be backed up for one run as `<logname>_prev.log`.

Conclusion

In this tutorial, you have:

1. Defined the necessary information in `design.tcl` to run synthesis and implementation on the top-level and OOC modules.
2. Defined the necessary physical, timing, and context constraints to implement an OOC design using the Top-Down Module Reuse flow.
3. Verified results using reports generated by the scripts, and by loading the top-level assembled design into the Vivado IDE.