# Vivado Design Suite Tutorial:

## *Hierarchical Design*

**UG946 (v 2013.3) October 30, 2013**

**Notice of Disclaimer**

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of the Limited Warranties which can be viewed at http://www.xilinx.com/warranty.htm; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in Critical Applications: http://www.xilinx.com/warranty.htm#critapps.

©Copyright 2012 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

# Revision History

The following table shows the revision history for this document.

| Date | Version | Changes |
|---|---|---|
| 10/30/2013 | 2013.3 | Initial Release |

# Table of Contents

Table of Contents ........ 3

Hierarchical Design Tutorial ........ 5

Overview ........ 5

Tutorial Design Description ........ 5

Software Requirements ........ 5

Hardware Requirements ........ 5

Locating Tutorial Design Files ........ 6

Lab 1: Top-Down Module Reuse ........ 7

Introduction ........ 7

Step 1: Running Bottom-up Synthesis ........ 7

Open the Provided Project ........ 7

Step 2: Constraining the In-Context Design ........ 9

Pblock Constraints ........ 9

HD.PARTPIN_RANGE Constraints ........ 10

Embedded I/O Constraints ........ 11

Step 3: Generating the Out-of-Context Constraints ........ 12

Run hd_floorplan to Generate the OOC XDC files ........ 13

Examine the Timing XDC Constraints ........ 15

Examine the Physical XDC Constraints ........ 16

Examine the optimization XDC constraints ........ 16

Overview of Interface Budget XDC Constraints ........ 17

Examine the PartPin Locations ........ 17

Step 4: Running the Non-Project Flow ........ 18

Edit design.tcl ........ 19

Examine Other Tcl Scripts ........ 21

Run the Non-Project Flow ........ 21

Step 5: Verifying Timing Results ........ 22

Open Checkpoint in the Vivado IDE ........ 22
Verify Timing Results ........ 23
Checking the Log Files ........ 25
Conclusion ........ 25

# Hierarchical Design Tutorial

## 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 2013.3 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:

   http://www.xilinx.com/support/documentation/

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 data should be used each time you start the tutorial.*

# Lab 1: Top-Down Module Reuse

## 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 define the necessary floorplan, timing, and context constraints to implement three modules out-of-context. You will then study and modify a set of provided Tcl scripts to setup the module (OOC) and top-level (Reuse) implementations. Finally, you will source the provided implementation scripts, run the flow, and examine the results.

## Step 1: Running Bottom-up Synthesis

The Top-Down Module Reuse flow uses an in-context design to create the necessary OOC constraints. These constraints consist of physical constraints, such as Pblock and lock constraints, as well as localized timing constraints. In addition to these, you will need to define context constraints to tell the tools how the OOC modules will connect in the context of the full design to improve the OOC implementation results. In this tutorial, we will use the Vivado IDE along with scripts to create these constraints.

### Open the Provided Project

From this project, you will run bottom-up synthesis of each OOC module. The results are merged in the top-level design, and the entire in-context design will be implemented to enable timing driven placement of partition pins. XDC constraint files are generated for each instance of the OOC modules. This in-context design already has the following constraints.

- Pblock constraints for each instance of an OOC module
- I/O constraints (package pin locations, I/O standard, etc.)
- Global buffer location constraints
- Timing constraints (I/O and clocking constraints relative to the top-level)

Open the provided floorplanning project.

1. Start the Vivado IDE.
2. Select **File > Open Project**.
3. Browse to this location:

   `<Extract_Dir>/Vivado/project_floorplan/project_floorplan.xpr`

4. Expand the design in the Hierarchy Source view, and locate the instances of **cpuEngine**, **usbEngine0**, and **usbEngine1**.
5. Right-click on the instance **cpuEngine**, and choose **Set As Out-Of-Context Module**.
6. Click **OK** on the dialog box to name the new Block File Set as **or1200_top**.

   Note the changed icon for **cpuEngine**, as well as module specific synthesis run in the Design Runs tab.

7. Repeat steps 5 and 6 for **usbEngine0**. Note that when **usbEngine0** is set as an OOC module, **usbEngine1** is also automatically set as well. This is because this is a module level setting, and the two instances share the same module (**usbf_top**).

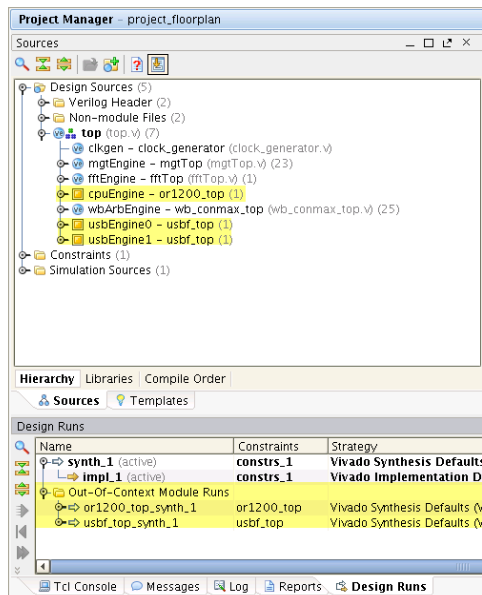



**Figure 1: Mark modules as Out-Of-Context**

8. Using Ctrl+ click, select both Out-of-Context Module runs (**or1200_top_synth_1** and **usbf_top_synth_1**) from the Design Runs window, then right-click and choose **Launch Runs**.

   If your machine has multiple processors available, you can run these two synthesis runs in parallel by setting the **Number of jobs** field in the Launch Selected Runs dialog box to 2.

9. After both OOC module synthesis runs have completed, the top-level synthesis run (**synth_1**) can be launched. Launch this active run by selecting **Run Synthesis** in the Flow Navigator.

The top-level synthesis cannot be run in parallel with the OOC module synthesis runs because no black box module definitions exist for these modules until their respective synthesis runs complete. After each OOC module synthesis completes, a black box stub file is automatically created by the tools. Alternatively, a custom black box module definition could be added to the project, which would allow all three synthesis runs to be launched in parallel (assuming 3 processors are available).

10. After **synth_1** finishes, a Synthesis Completed dialog box will open. Choose **Open Synthesized Design** and click **OK**.

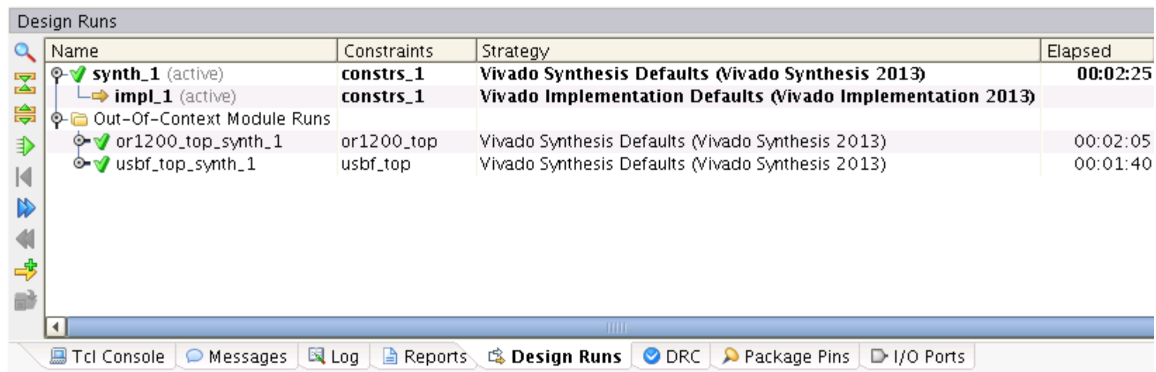



**Figure 2: Completed Synthesis Runs**

# Step 2: Constraining the In-Context Design

The full in-context design will be used to generate the constraints for the OOC implementations. Many 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, examine the existing top-level constraints, and add some additional constraints.

1. If not already open, open the synthesized design by selecting **Open Synthesized Design** from the Flow Navigator. This will combine the results of the top-level synthesis along with the OOC module synthesis runs to give a complete design.

## 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 add some additional

constraints to aid in the OOC implementation. For more information on using the Vivado IDE to create Pblock constraints please refer to the Vivado Getting Started Users Guide.

1. Click the Device window and examine the three Pblock rectangles. Note that each Pblock is on the hierarchical instance to be implemented OOC, and that rectangles do not overlap.
2. Open the XDC file, **top_flpn.xdc**, by double-clicking on it from the Sources window in the **Project Manager** view.

   Locate the Pblock constraints and note the syntax. The cells associated with the Pblock are an in-context hierarchical instance that will not exist when this module is implemented in the OOC flow. For the OOC implementation this will need to be modified as discussed in the **Create Physical XDC Files** section.
3. Edit `top_flpn.xdc` by adding the following CONTAIN_ROUTING constraints to the three Pblocks associated with the OOC modules.

   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]
   ```
4. Save `top_flpn.xdc` (**File > Save File**).

## 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. By default these PartPins are assigned to the slice range of the OOC module's Pblock.

However, in order to get high quality results from the OOC implementations, the placement of these ports can be guided by providing HD.PARTPIN_RANGE constraints. In this tutorial, the PartPins are ranged to the 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. Edit `top_flpn.xdc` by adding 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/*]
   ```

***TIP:*** *Each of the* `set_property` *constraints above must be on a single line in the constraints file. There must be no carriage returns placing a single constraint on two separate lines.*

2. Save `top_flpn.xdc` (**File > Save File**).

## 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 as 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. 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. 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 these constraints are applied to connect directly to OOC module pins.

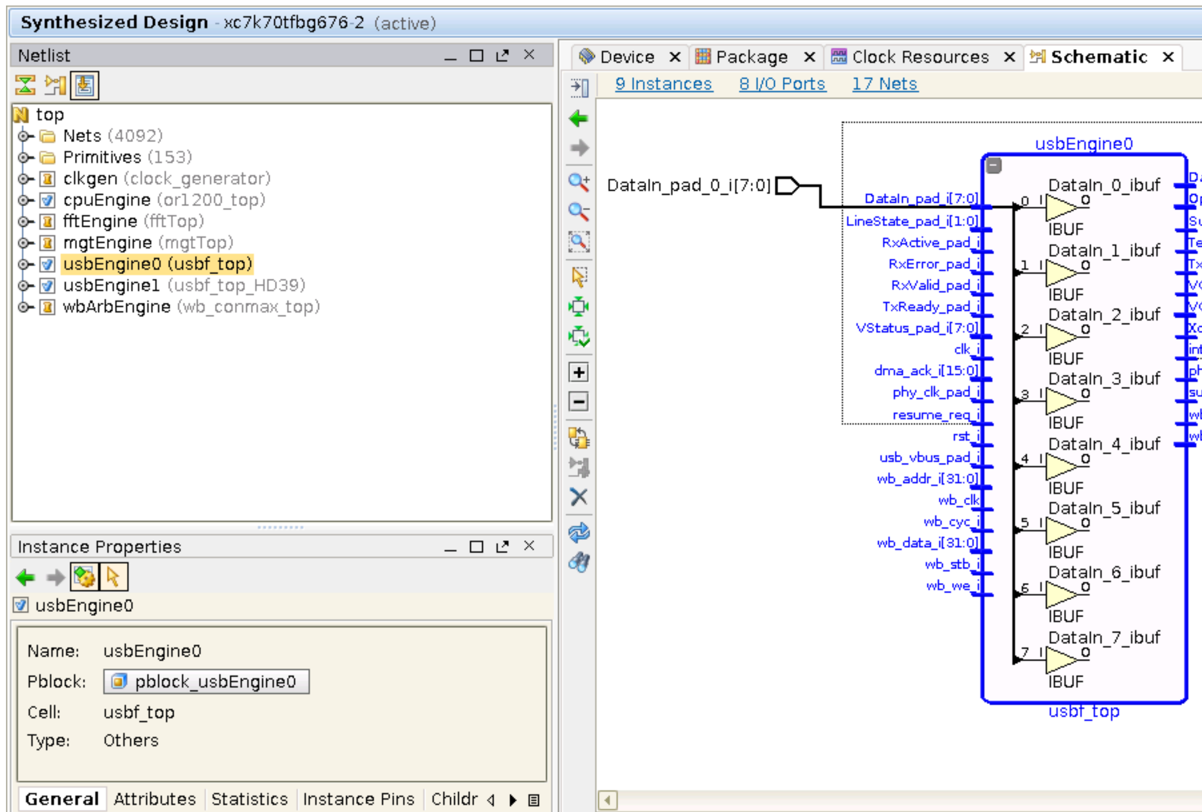



**Figure 3: Schematic View**

2. In the Synthesized Design view, select the instance **usbEngine0** from the Netlist window.
3. Press F4 to open the Schematic View (or alternatively select **Tools > Schematic**).
4. Select the input pin **DataIn_pad_i[7:0]** , and double-click on both sides of the pin to expand the schematic. Note that the IBUFs for this port are part of the **usbf_top** module, and not part of the top-level netlist.

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 Constraints section.

# Step 3: Generating the Out-of-Context Constraints

Using the in-context design, and the provided commands and scripts, all of the necessary constraints to run the OOC implementations can be automatically created. In this section you will source and run the `hd_floorplan` command, which 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.

Note the optimize and timing 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).

One additional XDC file will be generated by the implementation scripts during the OOC implementation.

- **`<instance>_ooc_budget.xdc`** - Interface budgeting constraints, set to 50% of the destination clock.

This XDC file contains budget constraints for the interface logic, and constrains input/output logic from/to the PartPins, respectively.

The following subsections describe what the `hd_floorplan` command does, and provides details about the constraints it creates.

## Run hd_floorplan to Generate the OOC XDC files

This Tcl proc is defined in a file provided with the tutorial.

1. Source the Tcl script `hd_floorplan_utils.tcl`. From the **Tcl Console** type the following command:

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

2. From the Netlist window of the Synthesized Design view, use Ctrl+click to select all the instances of the OOC Modules (**usbEngine0**, **usbEngine1**, and **cpuEngine**).

**CAUTION!** Make sure these instances are selected from the Netlist window of the open Synthesized Design view, and not the Sources window of the Project Manager view.

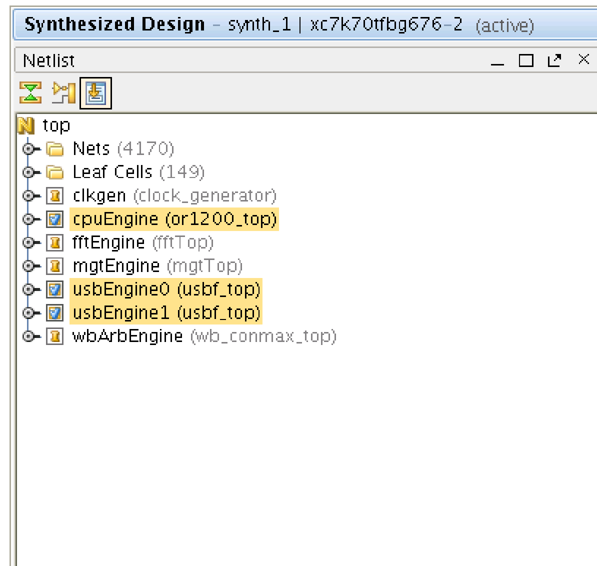



**Figure 4: Select all Instances of OOC Modules**

3. From the Tcl Console, run `hd_floorplan` on all selected instances.

```
hd_floorplan [get_selected_objects]
```

Running the above command will send all three instances to the `hd_floorplan` Tcl proc. This procedure calls several other procedures and other commands. Below is a list of commands done for each cell and instance.

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

## 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. In addition 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.

In order 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 `hd_floorplan` 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. 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.

The HD.PARTPIN_LOCS are a key constraint in controlling the OOC implementation. Using the in-context design we can run an in-context implementation to generate the HD.PARTPIN_LOCS. Specifically, we just need to run `place_design` on the in-context design. Prior to running the `write_xdc -cell` command the `hd_floorplan` Tcl proc runs `opt_design` and `place_design` on the in-context design to get the PartPin placement.

1. Open the physical XDC file generated by `hd_floorplan` 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 `hd_floorplan` 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 be 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 place close to the associated PartPin.

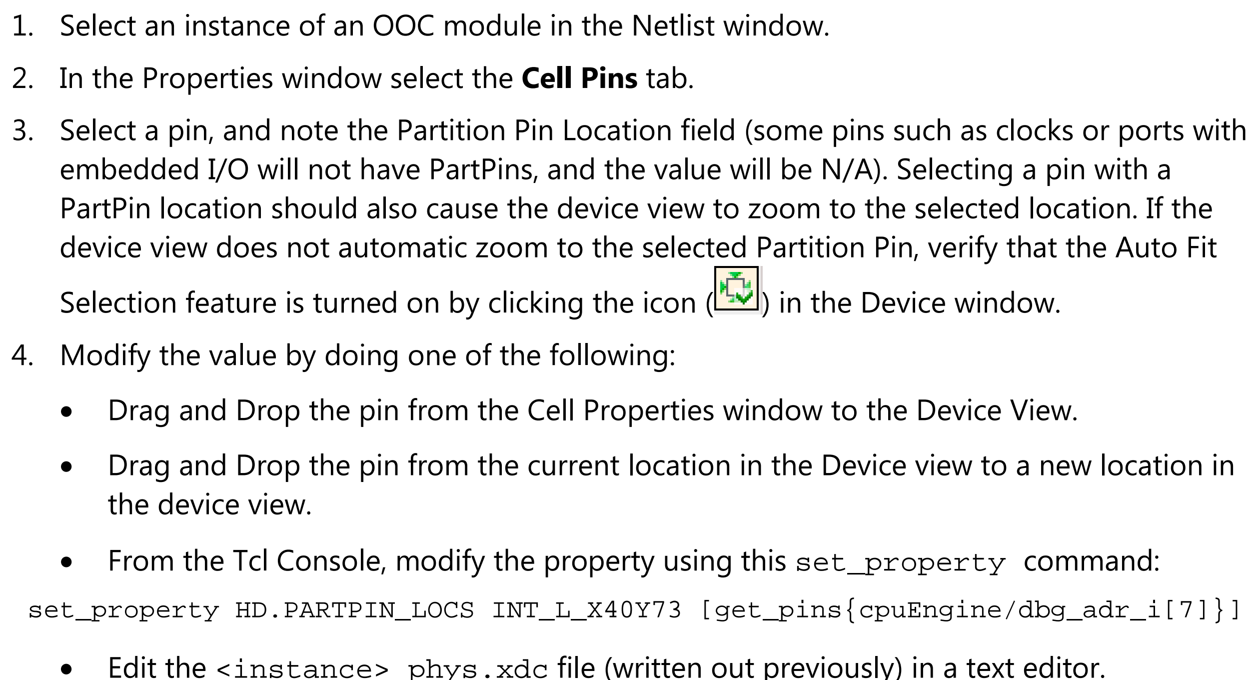
1. The interface budget XDC file will be generated during Step 4 of this lab. This file will be located at <Extract_Dir>/Sources/xdc/usbEngine1_ooc_budget.xdc.

   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. Additionally, 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 either the HD.PARTPIN_RANGE specified in the XDC, or by the module's associated Pblock range if no HD.PARTPIN_RANGE constraint exists.

The hd_floorplan procedure will highlight these locations with an incrementing color value. Highlighting these locations is not necessary, and is just done as a visual aid. The PartPin placement for this design is sufficient, but modifications to the placement could be done at this point if desired. These values can be modified using the following steps.



1. Select an instance of an OOC module in the Netlist window.
2. In the Properties window select the **Cell Pins** tab.
3. 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 icon ( ) in the Device window.
4. 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 this set_property command:

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

   - Edit the <instance>_phys.xdc file (written out previously) in a text editor.

5. Export modified PartPin location constraints using the `write_xdc -cell` command.

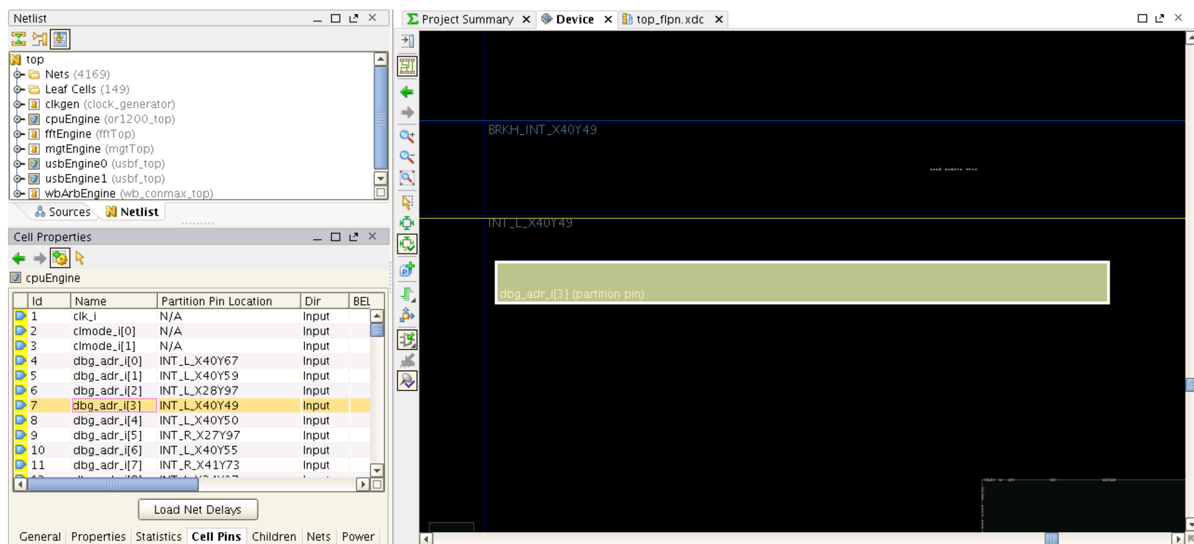



**Figure 5: PartPin Locations**

# Step 4: Running the Non-Project Flow

The current open project can be closed without saving any changes. All relevant information has been saved to the appropriate XDC files in the previous steps. 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 take you through setting up the `design.tcl` file for this tutorial design, and briefly talk about the other scripts and their interaction. For more information about the all of the provided scripts, please refer to the README.txt file included in the `<Extract_Dir>/Tcl` directory.

## Edit design.tcl

The intent of these scripts 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, top-down 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 only part of the flow is to be run (OOC synthesis or implementation, Top implementation, etc.).
3. The following section, `Top Definition`, must be defined. Only a few of the possible variables (`top_level`, `prj`, `vlog_headers`) need to be defined for this design. Other designs may require fewer or more module attributes to be modified. For more information on the available module attributes, see 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 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 be the comprehensive list of all files to be synthesized by Vivado synthesis.*

For this design, define the following values for the listed variables for top. Note some of these variables are already assigned for you. The lines that need to be added or modified are **bold**.

```
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.synth}

add_implementation $top
set_impl_attribute $top      top           $top
set_impl_attribute $top      implXDC       [list $xdcDir/${top}.xdc]
set_impl_attribute $top      impl          ${run.hdImpl}
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.*

4. The remaining sections define the module-level OOC synthesis and implementation runs. Again, only the required variables for the specific module need to be defined. For this design, define the following module properties for `usbf_top`.

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

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_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. The same holds true for the second module (or1200_top) and its OOC implementation (cpuEngine). Refer to the provided `design_complete.tcl` to resolve any issues in defining `design.tcl`.

## Examine Other Tcl Scripts

The rest of the Tcl files provided in the tutorial are generic to all designs. 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.

- **`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`.

## Run the Non-Project Flow

To run the provided scripts, and implement this design using the non-project flow, issue the following commands:

1. From a shell/prompt, **cd** to `<Extract_Dir>`.
2. Launch Vivado in batch mode and source `design.tcl` by typing:

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

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

# Step 5: Verifying Timing Results

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 Checkpoint in the Vivado IDE

As mentioned previously, a checkpoint can be loaded into memory for debug or analysis purposes. 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 with commands like `report_utilization` or `report_route_status`.
3. Start the IDE by typing **`start_gui`**.

   Note that 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 6 is an image of the final routed top design with the USB and CPU instances highlighted. Note how each OOC module is contained to its quadrant but the top-level logic is interspersed throughout the device.

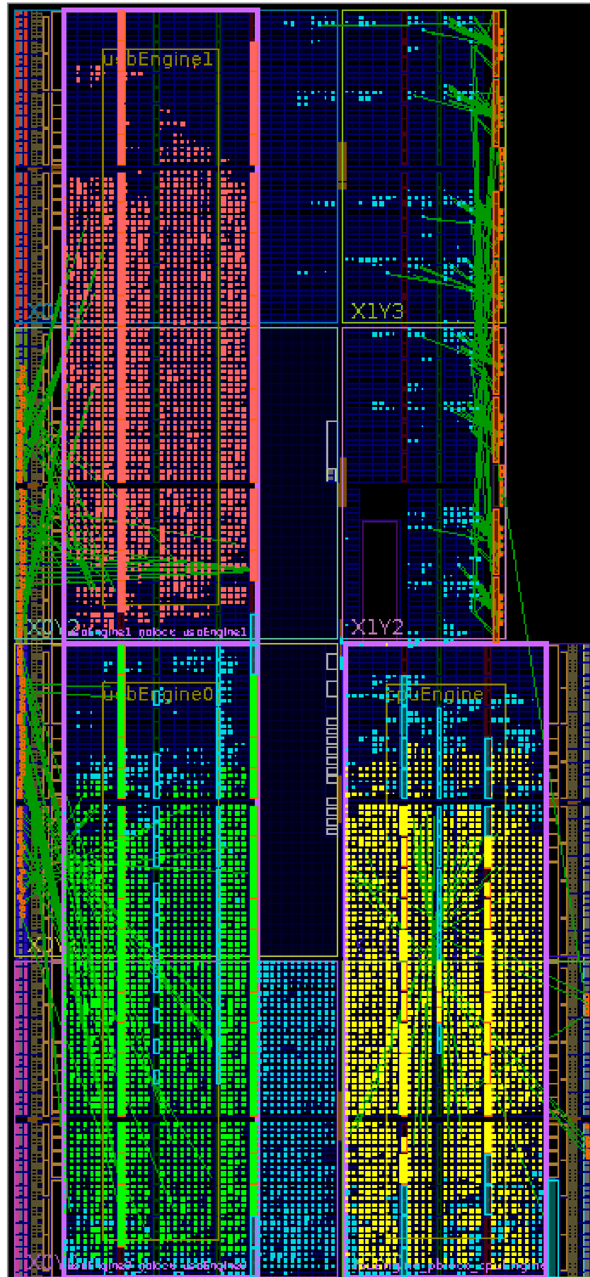



**Figure 6: Final Assembled Design**

## Verify Timing Results

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

   Verify timing interactively in the Vivado IDE by running `report_timing_summary` with a `-name` option.
4. With the Vivado IDE open and a post-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.

**Figure 7: 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 `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
   ```

2. 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
   ```

## Conclusion

In this tutorial, you have:

1. Defined the necessary physical, timing, and context constraints to implement a design using the Top-Down Module Reuse flow.
2. Set up the provided Tcl scripts to run 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 IDE.