Vivado Design Suite User Guide

Hierarchical Design

UG905 (2012.4) December 18, 2012

XILINX®

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	Revision
10/16/12	2012.3	Initial release.
12/18/12	2012.4	Revisions to manual for Vivado 2012.4 release. • In Introduction, added information that the "Reuse" flows should be considered Beta software in Vivado Design Suite 2012.4. • In Out-of-Context Constraints, removed this statement: "Only one rectangle per resource type is permitted for each Pblock in the current version of software". • In Limitations on Global Clock Routing, removed some limitations that applied to the previous release of the Vivado Design Suite. • Added information about the location of the *Vivado Design Suite Tutorial: Top-Down Reuse (UG946)*. See References in Appendix A.

Table of Contents

Revision History 2

Vivado Hierarchical Design

Introduction 4
Checkpoints 5
Design Considerations 5
Commands and Constraints 8
Tcl Scripts 17
Known Issues 21

Appendix A: Additional Resources

Xilinx Resources 22
Solution Centers 22
References 22

Vivado Hierarchical Design

Introduction

Hierarchical Design (HD) methodologies allow you to partition a design into smaller, more manageable blocks. In the Vivado Design Suite, these flows are based on the ability to implement a partitioned block out-of-context (OOC) from the rest of the design. The following is a list of the current methodologies in the Vivado Design Suite.

- **Module Implementation**: This flow implements a partitioned block or IP core out-of-context of the top level of the design in which it will be used. The module will be implemented in a specific part/package combination, and with a fixed location in the device. The OOC implementation results can be saved as a design checkpoint (DCP) file. This flow allows for the following use cases:
    - Implement a block independently from the rest of the design perform footprint and/or timing analysis. IO buffers are not inserted, but can be instantiated within the module.
    - Create a checkpoint for the module implementation so its results can be read into a full design implementation. To reuse the results, constraints to control the placement of module pins and interface logic must be used to ensure high quality results when read into a full design.
- **Module Reuse**: This flow uses the design checkpoint created from the Module Implementation flow within a top-level design. The use of this OOC module requires knowledge of where the module pins and interface logic have been placed so that the connecting logic can be floorplanned accordingly. The preservation level of the imported OOC module can be selected, but this only allows for minor placement and routing changes. This flow does not yet support moving or replicating the OOC implementation results to other areas of a device, or to a different device.
- **Top-Down Reuse**: This flow is similar to combining the Module Implementation and Module Reuse flows, with one key distinction. In this flow the top-level floorplanning (Pblocks, pinout, global logic, etc) is known, and is used to guide the OOC implementation. This allows for OOC module pin constraints, top-level input/output timing requirements, and boundary optimization constraints to all be created from the top-level design. Note that the OOC modules do not need to fully exist in this flow. Only a black box with port definition for any OOC modules is required to allow for synthesis of top alone, to prepare the context constraints needed for the OOC modules.

The Top-Down Reuse flow allows for the following use cases:

- Team Design enabling parallel synthesis and implementation of one or more modules within the design. Team members can implement their portions of a design independently, reusing their exact results in the assembled design.
- Implementation run time reduction by enabling the tools to only implement one module of the design, instead of the whole design. This can results in many more turns per day, and a reduced time to design, verify and meet timing on a per module basis.

In Vivado Design Suite 2012.4, the "Reuse" flows should be considered Beta software. Use of context constraints and thorough timing constraints are critical for high-quality results for clocks and inter-module paths. Further planning tools and designer assistance will be included in future versions of the Vivado Design Suite.

Other Hierarchical Design flows such as Partial Reconfiguration and the Isolation Design Flow will be supported in future versions of the Vivado Design Suite.

Checkpoints

In order to export and import results of a module implementation, Hierarchical Design flows use checkpoints. Checkpoints archive the logical design, physical design, and module constraints and are the only file needed to fully restore a design.

A saved checkpoint can only be read into the same part/package/speed grade combination in which it was originally generated.

RECOMMENDED: *The* **`-strict`** *option is recommended for HD flows to make sure that all data read in matches on the module interface exactly, ensuring a robust implementation. For more information on Checkpoints, please refer to the Vivado Design Suite User Guide: Implementation (UG904).*

Design Considerations

The Design Reuse methodology requires some special considerations to achieve optimal results. The following sections provide information to be considered when architecting, designing, and constraining a design for Vivado Hierarchical Design flows.

Optimization Limitations

Implementing a block out-of-context from the rest of the design prevents optimization that a design may normally see in a typical top-down flow. To limit the loss of performance due to these restrictions, follow these guidelines:

- Choose the block(s) to be implemented out-of-context carefully. Select blocks that are logically isolated from other logic in the design, and that can be physically constrained to a contiguous area of the device.
- Keep critical paths contained entirely within blocks, either in sub-modules or in top.
- Register boundaries between blocks to minimize losses due to module boundary optimization restrictions.
- Provide information on how the block will be used by defining context constraints. Some context constraints allow for additional optimization by telling the tools which ports are driven by constants, or are unconnected. See Out-of-Context Constraints in the Commands and Constraints section of this document.
- There is no optimization across module boundaries, so keep all related design elements partitioned together.

Floorplanning Requirements

Implementing a module out-of-context has the following requirements.

- Each module implementation must have a Pblock constraint to control the placement. If a Pblock is not used, placement conflicts will likely occur during the assembly phase.
- Pblock ranges for each OOC module must not overlap. If a top-level design is to import multiple OOC module results, the modules must occupy separated regions of the device.
- Nested or child Pblocks are supported within the OOC implementation as long as the nested Pblock range is fully contained by the parent Pblock range.
- All clock buffers (both in top and the OOC module) must be locked. Buffers inside of the OOC module must have LOC constraints, and locations of buffers in the top level should be identified by **HD.CLK_SRC** constraints.
- If the OOC implementation results are to be reused, the OOC module pins should be locked down during the OOC implementation using **HD.PARTPIN_RANGE** or **HD.PARTPIN_LOCS** constraints.

Hierarchy Considerations

Design Hierarchy is an important consideration. Where a design is partitioned can have significant effects on the quality of the results, and in some cases hierarchy may need to be added or modified to group appropriate modules together for an out-of-context implementation.

Dedicated Connections

It is recommend, and in some cases required, to keep components with dedicated connection is the same partition of the design. Having a dedicated connection span the boundary of an OOC module can cause reduced performance and/or implementation errors. The following is a list of components with dedicated connections.

- **IOLOGIC and IOBUF** - This includes connections from registers placed in the ILOGIC or OLOGIC, IDDR, ODDR, ISERDES, and OSERDES to IO components including IBUF, OBUF, IBUFDS, OBUFDS, IOBUF, and IOBUFDS.
- **GT components** - GTX, GTP, and their dedicated I/O connections.

Avoid placing any IO components that connect to each other in different partitions of a design.

IO and Clock Buffers

IO and clock buffers are supported inside of OOC modules. However, some special considerations should be taken into account depending on the use.

- **IO Buffers** - If an OOC port connects directly to an IO buffer in the top-level, it is recommended to move this buffer inside of the OOC module for better results - implementation tools will have full visibility to all IO components and place them most efficiently. Of course this is not possible in all situations (e.g. if an OOC port connects directly an IBUF in the top-level, but that IBUF also drives other logic not in the OOC module), and in those cases the logic inside of the OOC module should be controlled with an HD.PARTPIN_LOCS constraint. Refer to the Command and Constraints section for more information.
- **Regional Clock Buffers** - If a BUFR is within the OOC module it should be locked down to a specific location. The tools will then appropriately place logic driven by the BUFR. However, if the BUFR is in the top-level design, and the OOC Pblock spans more clock regions than the BUFR has access to, then more information must be supplied. A nested Pblock must be created with a range that is a subset of the range defined by the OOC Pblock. The nested Pblock would contain all of the cells driven by the BUFR, and can be created with the following commands:

  - `create_pblock -parent <parent_pblock_name> <nested_pblock_name>`
  - `add_cells_to_pblock <nested_pblock_name> [get_cells -of [get_nets -of [get_ports <bufr_clock_port>]]]`
  - `resize_pblock <nested_pblock_name> -add {SLICE_Xx1Yy1:SLICE_Xx2Yy2}`

  This should be done for each module port that is driven by a BUFR in the top-level. The range for the nested Pblock must correspond to the BUFR location in the top-level implementation. A mismatch between BUFR location at the top level and the corresponding Pblock range for the OOC module can lead to unroutable conditions during top-level implementation.

- **Global Clock Buffers** - Global buffers are supported inside of an OOC module. When a BUFG is inside of an OOC instance the clock net will be routed on global routing in the OOC implementation. If an OOC port is driven by a clock net in the top-level, the clock net will not be routed during the OOC implementation, and timing estimations will be used to determine clock delays/skew. The HD.CLK_SRC constraint should be used to help improve timing estimations in this case.

Commands and Constraints

The HD flows are currently only supported through the project-less batch/Tcl interface (no UI or project based commands). Example scripts are provided in the *Vivado Design Suite Tutorial: Top-Down Reuse (UG946)*, along with step by step instructions for setting up the flows.

The following sections describe a few specialized commands and/or options needed for the HD flows. Examples of how to use these commands to run an HD flow are given. For more information on individual commands please refer to the Tcl chapter of the Vivado User Guide.

Out-of-Context Commands

Synthesizing or implementing a block out of context requires that the tools be run in an "out-of-context" mode. Other than that, the commands used to run an out-of-context flow are the same as for any other flow. There are currently no unsupported commands for synthesis, optimization or implementation.

Synthesis

There are several supported synthesis tools and methods supported for this flow. The following is a list of supported tools.

- **XST**: Bottom-up synthesis or Incremental synthesis using partitions (PXML file)
- **Synplify**: Bottom-up synthesis or Compile Points (using Hierarchical Projects to generate individual netlists)
- **Vivado Synthesis**: Bottom-up only

IMPORTANT: *Bottom-up synthesis refers to a synthesis flow where each block has its own synthesis project. This generally involves turning off automatic IO buffer insertion for the lower level blocks.*

This document only covers the Vivado Synthesis flow. For information on the other flows please refer to the XST User Guide, or Synopsys Synplify documentation.

Vivado synthesis for this flow will be run in batch mode using the **synth_design** command:

```
synth_design -mode out_of_context -flatten_hierarchy rebuilt -top
<top_module_name> -part <part>
```

Table 1: **synth_design Options**

Command Option	Description
-mode out_of_context	Prevents IO insertion for synthesis and downstream tools. The mode will be saved in checkpoints if **write_checkpoint** is issued.
-flatten_hierarchy rebuilt	There are several values allowed for **-flatten_hierarchy**, but **rebuilt** is the recommended setting for HD flows.
-top	This is the module/entity name of the module being synthesized. This switch can be omitted if **set_property top <*top_module_name*> [current_fileset]** is issued prior to **synth_design**.
-part	This is the Xilinx part being targeted (e.g., **xc7k325tffg900-3)**

The **synth_design** command synthesizes the design and stores the results in memory. In order to write the results out to a file, a command must be issued. The recommend formats to write out are either EDIF or a Vivado checkpoint. Either of the following commands can be used.

```
write_edif <file_name>.edf

write_checkpoint <file_name>.dcp
```

If you would like to be able to close the project-less design and run implementation without having to rerun synthesis, you must use one of the above commands to write the synthesis results to a file.

Implementation

This section describes the necessary commands to implement a module instance in an out-of-context flow. Note that if this module has multiple instances instantiated in a top level design and the Assembly Flow is going to be used, multiple OOC implementations each with unique Pblock constraints are required to generate the necessary implementation results.

If **synth_design -mode out_of_context** was previously run, and the results are still in memory, then implementation can be run directly. For example, the following implementation commands can be used.

- **read_xdc** (if all constraints are not already loaded)
- **opt_design** (optional)
- **place_design**
- **phys_opt_design** (optional)
- **route_design**

If there is currently no design in memory, then a design must be loaded. This can be done in one of two ways.

Method 1: Read Netlist Design

```
read_edif <file_name>.edf/edn/ngc
link_design -mode out_of_context -top <top_module_name> -part <part>
```

Table 2: **link_design Options**

Command Option	Description
-mode out_of_context	Load a netlist design in an out-of-context mode. Enables special checking and optimization for downstream tools.
-part	This is the Xilinx part being targeted (e.g., **xc7k325tffg900-3**)
-top	This is the module/entity name of the module being implemented. This switch can be omitted if **set_property top** ***<top_module_name>*** **[current_fileset]** is issued prior to **link_design**.

If the **-mode out_of_context** option is not issued for **link_design** after reading the design netlist(s), subsequent implementation steps will treat the design as a full design and trim any sourceless or loadless signals. The OOC mode must be defined during either **synth_design** or **link_design** to run the Module Implementation flow.

Method 2: Read Checkpoint

```
read_checkpoint <file_name>.dcp
```

IMPORTANT: *The* **`read_checkpoint`** *command does not have a* **`-mode out_of_context`** *option. The mode is saved as part of the checkpoint, therefore it is critical to make sure that the tools are in the correct mode when writing out a checkpoint.*

Out-of-Context Constraints

In order to successfully implement a module out-of-context, it is required to describe the context in which the OOC module will be instantiated in the full design. This includes definition of clock ports, constants and/or unconnected pins, physical constraints, and timing requirements.

The following tables are a list of timing, placement, and context constraints that should be used for an out-of-context implementation. Many of these constraints are used for any design flow, and more information can be found in the *Vivado Design Suite User Guide: Using Constraints (UG903)*.

Table 3: **Timing Constraints**

Constraint Name	Description
`set_input_delay`	Used to define input delays into to budget the time allowed for the OOC module. Will help control placement in the OOC implementation and ease timing closure at the top level.
`set_output_delay`	Used to define output delays into to budget the time allowed for the OOC module. Will help control placement in the OOC implementation and ease timing closure at the top level.
`create_clock`	Used to define clocks on the OOC module ports. A **`create_clock`** constraint should exist for every clock port, whether the clock buffer is instantiated in the top level or the OOC module.

Timing Constraint Examples:

- **`create_clock -period 8.000 -name clk -waveform {0.000 4.000} [get_ports clk]`**
- **`set_input_delay -clock <clock_name> 3.0 [get_ports <ports>]`**
- **`set_output_delay 5.0 -clock [get_clocks <clock_name>] [get_ports]`**

These timing constraints are scoped to the OOC module itself. The OOC implementation should constrain all timing into, out of, and internal to the instance. This includes special case paths such as false paths and multi-cycle paths.

Table 4: **Pblock Commands and Properties**

Command/Property Name	Description
create_pblock	Command used to create the initial Pblock for each OOC instance.
resize_pblock	Command used to define the site types (SLICE, RAMB36, etc.) and site locations that will be owned by the Pblock.
add_cells_to_pblock	Command used to specify the instances that will belong to the Pblock. This is typically a level of hierarchy as opposed to individual instances. For OOC implementations, **-top** can be used to specify all cells under the OOC module instead of specifying cell names.
CONTAIN_ROUTING	Pblock property used to control the routing to prevent usage of routing resources not owned by the Pblock. Default value is false. Only paths that are entirely owned by the Pblock RANGE will be contained (e.g., If no BUFGMUX range exists, paths from/to a BUFGMUX will not be contained. This is the desired behavior for many components like a BUFGMUX).
EXCLUDE_PLACEMENT	Pblock property used to prevent the placement of any logic, not belonging to the Pblock, inside the defined Pblock RANGE. The default value is **false**. This property has no effect on the OOC implementation, but will affect the placement of the top-level logic during assembly. Xilinx recommends you leave this as **false** for the best results during assembly.

Pblock Command and Property Examples:

- **create_pblock <*pblock_name*>**
- **add_cells_to_pblock [get_pblocks <*pblock_name*>] -top**
- **resize_pblock [get_pblocks <*pblock_name*>] -add {SLICE_X0Y0:SLICE_X100Y100}**
- **resize_pblock [get_pblocks <*pblock_name*>] -add {RAMB18_X0Y0:RAMB18_X2Y20}**
- **set_property CONTAIN_ROUTING true [get_pblocks <*pblock_name*>]**

Note that the cells added to the Pblock are specified using **-top**. This is because in an out-of-context implementation the OOC instance will be the top-level, and the entire OOC instance must be contained by the Pblock. Using **-top** also allows the Pblock to be properly translated to the correct level of hierarchy when the OOC module is imported into the top level design.

Nested Pblocks are permitted for floorplanning logic within an OOC module. Any child Pblock must be completely contained within the parent. The parent-child relationship between Pblocks is declared using the -parent switch, as shown here:

```
create_pblock -parent <parent_pblock_name> <child_pblock_name>
```

In addition to the timing and physical constraints shown above, there are constraints to define context for an OOC implementation. Context constraints define the environment of the top-level into which the OOC implementation will be imported.

Table 5: **Context Commands and Properties**

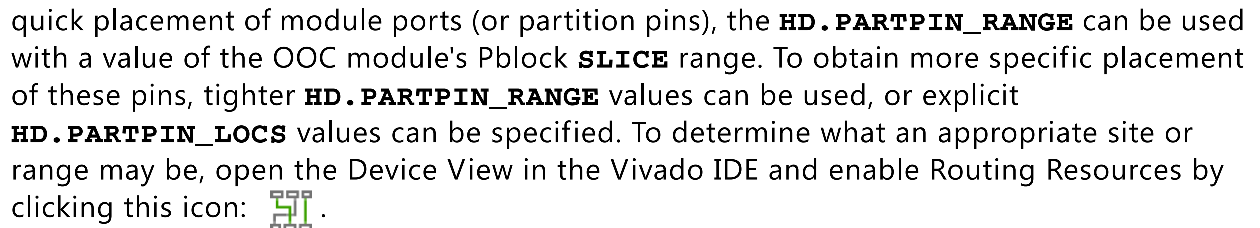
Command/Property Name	Description
HD.CLK_SRC	Used in the OOC implementation to tell the implementation tools if a clock buffer will be used outside of the out-of-context module. The value is the location of the clock buffer instance.
HD.PARTPIN_LOCS	Used to define a specific interconnect location for the specified port to be routed. Overrides an **HD.PARTPIN_RANGE** value. Affects placement and routing of internal OOC logic. Do not use on clock ports, as this will assume local routing for the clock, and do not use on dedicated connections.
HD.PARTPIN_RANGE	Used to define a range of component sites (SLICE, DSP, BRAM) or interconnect tiles (INT) that can be used to route the specified port(s). Do not use on clock ports, as this will assume local routing for the clock. Do not use on dedicated connections.
set_logic_unconnected	Allows for additional optimization for any specified output ports that will be left unconnected in Top.
set_logic_one	Allows for additional optimization for any specified input ports that are driven by VCC in Top.
set_logic_zero	Allows for additional optimization for any specified input ports that are driven by GND in Top.

IMPORTANT: *Incorrectly specifying the* **set_logic** *boundary optimization constraints can lead to incorrect behavior and tool errors. For example, defining an output port as unconnected in the OOC module that is actually used in the top-level can lead to errors such as:* `ERROR:[Opt 31-67] Problem: A LUT2 cell in the design is missing a connection on input pin I0, which is used by the LUT equation.`

Context Constraint Examples:

- **set_property HD.CLK_SRC BUFGCTRL_X0Y16 [get_ports <*port_name*>]**
- **set_property HD.PARTPIN_LOCS INT_R_Xx1Yy1 [get_ports <*port_name*>]**
- **set_property HD.PARTPIN_RANGE SLICE_Xx1Yy1 :SLICE_Xx2Yy2 [get_ports <*port_name*>]**
- **set_logic_unconnected [get_ports <*port_name*>]**
- **set_logic_one [get_ports <*port_name*>]**
- **set_logic_zero [get_ports <*port_name*>]**

By default, in the Module Implementation flow, interface nets (nets inside the module connected to the OOC module ports) will not be routed. To have these interface nets routed, it is required to lock the module ports using **HD.PARTPIN** constraints. To get a

quick placement of module ports (or partition pins), the **HD.PARTPIN_RANGE** can be used with a value of the OOC module's Pblock **SLICE** range. To obtain more specific placement of these pins, tighter **HD.PARTPIN_RANGE** values can be used, or explicit **HD.PARTPIN_LOCS** values can be specified. To determine what an appropriate site or range may be, open the Device View in the Vivado IDE and enable Routing Resources by clicking this icon:  .

When you zoom in, you'll see INT locations as shown in Figure 1 (routing resources are hidden to simplify this image):

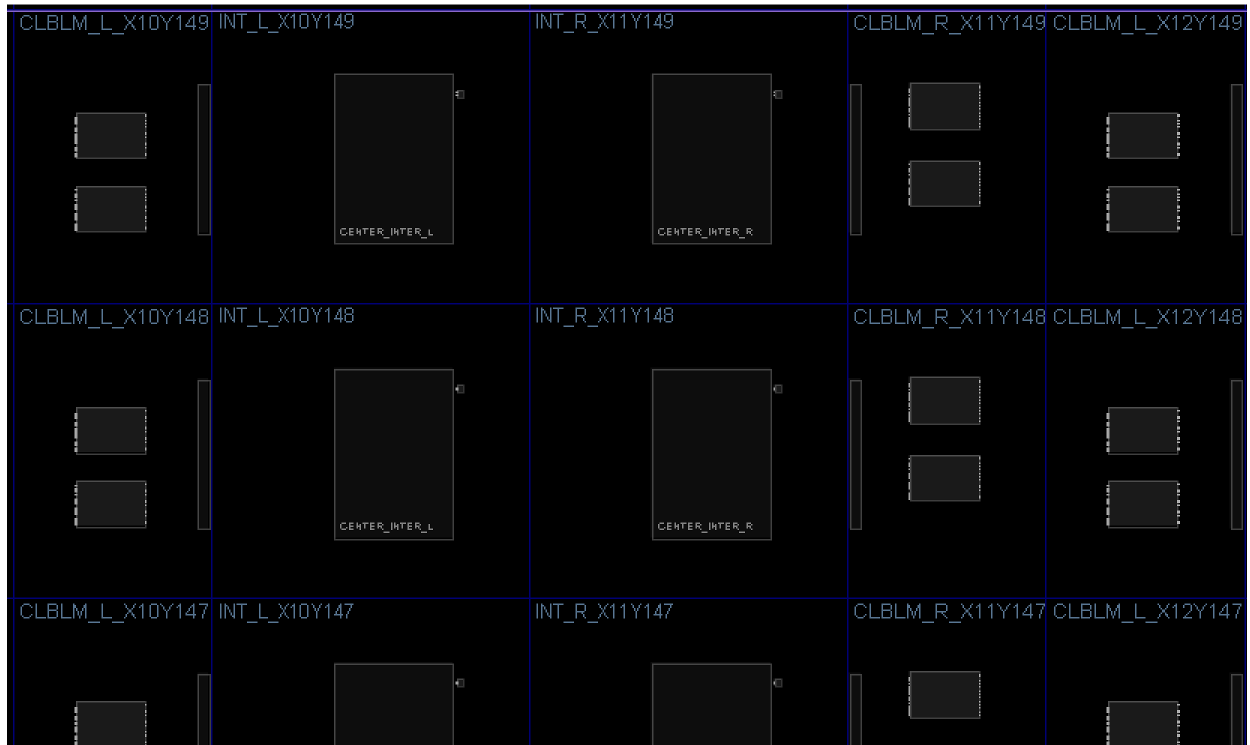


Figure 1: **INT Tile Locations**

Top-Level Assembly (Reuse) Commands

Assembly requires the tools to read in a previously implemented module from the Module Implementation flow. A checkpoint for each partitioned instance must exist, and each is read into a top level design with a black box for each instance that will load an implemented OOC module. The standard implementation commands are then used to implement the portions of the design that are not already placed and routed (i.e. Top).

Synthesis

You must have a top-level netlist with a black box for each partitioned instance. This requires the top-level synthesis to have module/entity declarations for the partitioned instances, but no logic.

The top level synthesis will typically infer IO buffers on all top level ports. However, if IO buffers are specifically instantiated in an OOC module, you must turn off IO buffer insertion in the top-level synthesis on a port-by-port basis. For Vivado synthesis, the attribute to do this is **BUFFER_TYPE**. For more information on **BUFFER_TYPE** and other synthesis attributes, please refer to the *Vivado Design Suite User Guide: Synthesis (UG901)*.

Implementation

Top level implementation is done the same as it is for standard designs, except for the addition of the following two steps.

1. Read in an OOC checkpoint for each partitioned instance.
2. Choose the level of preservation to maintain (logical, placement, or routing).

Read in OOC checkpoint

Reading in an OOC checkpoint is done using the **read_checkpoint** command. The top level design must already be opened, and must have a black box for each partitioned instance.

```
read_checkpoint -cell <cell name> <file> [-strict]
```

Table 6: **read_checkpoint Switches**

Switch Name	Description
-cell	Used to specify the full hierarchical name of the OOC module.
-strict	Requires exact ports match for replacing cell, and checks that part, package, and speed grade values are identical.
<*file*>	Specifies the OOC checkpoint to be read in.

Setting Preservation Level

After reading in the OOC checkpoint, the preservation level for this module must be defined.

To lock the placement and/or routing of an imported OOC checkpoint, a Tcl proc called **lock_design** exists and is provided with the *Vivado Design Suite Tutorial: Top-Down Reuse (UG946)*. Please refer to the tutorial for example usage.

```
lock_design <cell_name> <preservation> <lock_value>
```

Within this proc, the following preservation levels are available:

- **Logical** - Preserves the logical design. Any placement or routing information is still used, but can be changed if the tools can achieve better results.
- **Placement** (default) - Preserves the logical and placed design. Any routing information is still used, but can be changed if the tools can achieve better results.
- **Routing** - Preserves the logical, placed and routed design. Internal routes are preserved, but interface nets are not. Requires the use of the **CONTAIN_ROUTING** constraint during the OOC implementation.

No other combination of these properties is supported. Also, note that regardless of the desired preservation level, the entire physical database will still be read in (including routing), and will not be modified unless the tools determine that better results can be obtained.

Table 7: **Tcl proc lock_design arguments**

Argument Name	Description
<cell_name>	This is the hierarchical cell name to be locked.
<preservation>	Specifies the preservation level. Values are **logical**, **placement**, or **routing**. The default value is **placement**.
<lock_value>	Values are **1** to lock, and **0** to unlock.

Top-Level Assembly (Reuse) Constraints

When reusing an out-of-context module in a top-level design, all normal design constraints can be applied. The out-of-context constraints used in the out-of-context implementation will be saved in the checkpoint, and will applied (where applicable) to the design as well.

Tcl Scripts

Scripts are provided to run this flow in the *Vivado Design Suite Tutorial: Top-Down Reuse (UG946)*. This section describes the details of the provided scripts, and is intended as a reference as you modify the scripts to meet your designs needs.

design.tcl

The script `design.tcl` is the primary file that you need to be familiar with to run the provided scripts. Later sections will talk about the various other scripts that are provided, but it is possible to setup a design and the flow using these scripts by only editing design.tcl.

Section 1 - Tcl Variables

The commands in this section just set up the location of where the other Tcl files exist. This does not need to be edited. There is a command that is commented out to set one or more Tcl params. This is typically not needed, but the scripts do support this if necessary.

```
set tclParams [list <param1> <value> <param2> <value> ... <paramN> <value>]
```

Section 2 - Setup Variables

The variables defined in this section define which parts of the flow to run. This will allow you to focus one stage of the design, and only run other steps when necessary. The table below describes these variables in detail.

Table 8: **Flow Control Variables**

Variable Name	Description
synthBlock	If set to one, this will run **synth_design -mode out_of_context** for all modules defined by $oocModules. Output will be located at `$synthDir/$module`, and it also copied to `$netlistDir/$module`.
implBlock	If set to one, this will run implementation (**opt_design**, **place_design**, **phys_opt_design**, **route_design**) for all $module#_instances defined in $oocModules. To control which parts of implementation are run, please refer to the run.tcl script. Netlists must exist at `$netlistDir/$module`. Either manually add netlist to this folder for 3rd party synthesis flows, or run **synthBlock** prior to this step.

Table 8: **Flow Control Variables**

Variable Name	Description
synthTop	If set to one, this will run synth_design for the top-level module. Output will be located at `$synthDir/$top`, and it also copied to `$netlistDir/$top`.
implTop	If set to one, this will load the top-level netlist, load and lock down preservation levels for each OOC instance, and run implementation (**opt_design**, **place_design**, **phys_opt_design**, **route_design**) for the top level. To control which parts of implementation are run, please refer to the run.tcl script. Each OOC instance must have a checkpoint at `$implDir/$instance`. If this does not exist, run **implBlock** prior to this step.

In addition to the flow control variables, this section also contains input and output directory locations. The table below describes these locations.

Table 9: **Directory Variables**

Variable Name	Description
srcDir	This is the location of primary source directory containing subdirectories for `rtl`, `prj`, `xdc`, and `netlist`.
rtlDir	Assumed to be a subdirectory of `$srcDir`. This directory will contain all of the RTL source files for synthesis. It is recommended to have subfolders for each module in this directory (ie. $top, $module1, $module2).
prjDir	Assumed to be a subdirectory of `$srcDir`. This directory is for XST `prj` files. This helps in converting designs from XST to VDS, but a PRJ can also be easily created if one does not exist. The `prj` will be parsed and tell the tools which files to read in for synthesis. For an example of syntax, please reference the files located in the `./Sources/prj` directory of the *Vivado Design Suite Tutorial: Top-Down Reuse (UG946)*.
xdcDir	Assumed to be a subdirectory of `$srcDir`. This is the location of the top and OOC instance XDC files.
netlistDir	Assumed to be a subdirectory of `$srcDir`. This is the location of netlist files for 3rd party synthesis, as well as the location for **synth_design** results when using the VDS flow.
synthDir	This is the output for **synth_design** when running an RTL flow. Subdirectories for the top-level and each OOC module will be created. These results get copied to `$netlistDir` to allow for compatible implementation scripts regardless of the synthesis flow used.
implDir	This is the output for all implementation. Subdirectories for the top-level and each OOC instance will be created.

Section 3 - Top Definition

This section defines everything about the top-level module for synthesis and/or implementation.

Table 10: **Top Module Variables**

Variable Name	Description
top	Name of top level module.
top_prj	Path to PRJ file for synthesis. If defined, this will override and values of **top_vlog**, **top_sysvlog**, and **top_vhdl**.
top_vlog_headers	Used to identify any Verilog Header files.
top_vlog_defines	Used to define any Verilog definitions.
top_vlog	Used to define Verilog files and the associated compile library.
top_sysvlog	Used to define System Verilog files and the associated compile library.
top_vhdl	Used to define VHDL files and the associated compile library.
top_cores	Used to define any IP core netlists.
top_xdc	XDC file to be used for synthesis.
top_impl_xdc	XDC file to be used for implementation.
top_impl_preservation	Do not edit. Currently not supported.
top_impl	Do not edit. A list of all variables for the top-level implementation.
topModule	Do not edit. A list of all variables related to the top-level.

Note that for 3rd party synthesis flows, all Verilog and VHDL variables can be left as empty. The netlist for the top-level should be located at `$netlistDir/$top/$top.{edf, edn, ngc}`.

Section 4 - OOC Module Definition

This section defines everything about an OOC module, including all of its instances in the top-level design. This section must be replicated for every OOC module in the design. In addition, the variables name **module#_inst#_*** must be replicated for every instance of the module.

Table 11: **OOC Module Variables**

Variable Name	Description
module#	Name of module
module#_prj	Path to PRJ file for top-level synthesis. If defined, this will override and values of **top_vlog**, **top_sysvlog**, and **top_vhdl**.
module#_vlog_headers	Used to identify any Verilog Header files.
module#_vlog_defines	Used to define any Verilog definitions.
module#_vlog	Used to define Verilog files and the associated compile library.

Table 11: **OOC Module Variables**

Variable Name	Description
module#_sysvlog	Used to define System Verilog files and the associated compile library.
module#_vhdl	Used to define VHDL files and the associated compile library.
module#_cores	Used to define any IP core netlists.
module#_xdc	XDC file to be used for synthesis.
module#_inst#_name	Local instance name of the OOC module (for example, u1).
module#_inst#_hierName	Full hierarchical instance name of the OOC module (for example, a/b/u1). Used to correctly import the OOC instance into the top-level during assembly.
module#_inst#_xdc	XDC file to be used for the OOC implementation.
module#_inst#_preservation	Used to define the preservation of the imported OOC instance during assembly (logical, placement, routing). Passed to the lock_design proc.
module#_instance#	Do not edit. A list of all variables about the OOC instance.
module#_instances	A list of all **$module#_instance#** variables defined for the module.

TIP: *For 3rd party synthesis flows or netlist-only IP, all Verilog and VHDL variables can be left as empty. The netlist for the module should be located at `$netlistDir/$module#/$module#.{edf, edn, ngc}`. Any lower level IP core netlists for the module can be defined by **$module#_cores**.*

Section 5 - Design Summary

This section contains only a couple lists that must be a comprehensive list of all modules to be synthesized and/or implemented by the scripts. The **$Modules** variable should list all OOC modules listed in **$oocModules** as well as **$topModule**.

This section also sources the `run.tcl` script that will call the synthesis and implementation tools.

Known Issues

This section reports a few known issues with the current 2012.4 release for Hierarchical Design flows.

Limitations on IO logic in OOC Module

It is currently not recommend to insert IO buffers within the OOC module. If there is an IP core with IO or other situation that requires this, put all dedicated type connections in the same partition. If an OOC module has I/OLOGIC blocks, then its associated buffer should be in the module as well.

Limitations on Routing Preservation

The following limitations currently exist with respect to routing preservation when importing an OOC module.

- Interface nets are not preserved.
- The OOC implementation must have used the **CONTAIN_ROUTING** Pblock property. This is to prevent routing conflicts that cannot be resolved once the routing is locked.
- Interface routes must exist. Even though they are not preserved, the interface routes must exist and must have been routing during the OOC implementation. This requires the use of **HD.PARTPIN** constraints in the OOC implementation.

Limitations on Global Clock Routing

Clocks driven by a buffer in the top level will not be routed during the OOC implementation. Routing estimations will be used, and the use of HD.CLK_SRC will help improve routing estimates. Clock buffers within the OOC module will be routed during OOC implementation.

Known Issue for Timing Constraints Processing

All physical and timing constraints are stored in the design XDC that is created by **write_checkpoint** or **write_xdc**. In order to avoid timing constraint duplication or other issues when a reuse flow is used, it is advised to remove boundary timing constraints when creating these OOC module checkpoints. These constraints include **create_clock**, **set_input_delay**, and **set_output_delay** for clocks created at the top level. These constraints can be cleared by issuing the **reset_timing** command before writing the module checkpoint. This will clear out internal timing constraints as well as **set_logic_*** constraints, so these must be re-established in order for the top level design implementation is done. This can be achieved by re-reading a module XDC containing these constraints only, or by including these constraints in the top-level XDC.

Appendix A

Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx® Support website at:

www.xilinx.com/support.

For a glossary of technical terms used in Xilinx documentation, see:

www.xilinx.com/company/terms.htm.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

References

- Vivado™ Design Suite 2012.4 Documentation (http://www.xilinx.com/support/documentation/dt_vivado_vivado2012-4.htm)
- *Vivado Design Suite Tutorial: Top-Down Reuse (UG946)*
  This document is available upon request (see Answer Record 52794).
- *Hierarchical Design Methodology Guide (UG748)*