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 — 2013 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.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/30/13</td>
<td>2013.3</td>
<td>Initial Xilinx release.</td>
</tr>
<tr>
<td>12/18/13</td>
<td>2013.4</td>
<td>Revisions to manual for Vivado Design Suite 2013.4 release:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In Floorplan in the Vivado IDE in Chapter 2, added information about running</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Partial Reconfiguration DRCs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added a section about Reconfigurable Partition Interfaces in Chapter 3.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added a section about Building Up Implementation Requirements in Chapter 3.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In Verifying Configurations in Chapter 2, described new options to pr_verify command.</td>
</tr>
</tbody>
</table>
# Table of Contents

Revision History ......................................................... 2

Chapter 1: Introduction
  Overview ........................................................................... 5
  Introduction to Partial Reconfiguration ............................... 6
  Terminology ....................................................................... 7
  Design Considerations ...................................................... 9

Chapter 2: Vivado Software Flow
  Software Flow Overview ................................................... 13
  Partial Reconfiguration Commands ..................................... 14
  Partial Reconfiguration Constraints and Properties ................. 19
  Software Flow .................................................................... 27
  Tcl Scripts .......................................................................... 31

Chapter 3: Design Considerations and Guidelines
  Introduction ....................................................................... 32
  Design Hierarchy ............................................................. 32
  Global Clocking Rules ...................................................... 35
  Partition Pin Placement ..................................................... 37
  Active Low Resets and Clock Enables .................................. 37
  Decoupling Functionality .................................................. 38
  Reconfigurable Partition Pblock Sizes and Shapes ................. 39
  Black Boxes ...................................................................... 42
  Implementation Strategies ............................................... 43
  Design Revision Checks ................................................... 45
  Simulation and Verification ............................................... 45

Chapter 4: Configuring the FPGA Device
  Configuration Overview .................................................... 46
  Configuration Modes ....................................................... 47
  Downloading a Full Bit File .............................................. 48
  Downloading a Partial Bit File ............................................ 49
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Design for Configuring an FPGA Device</td>
<td>50</td>
</tr>
<tr>
<td>Partial Bitstream CRC Checking</td>
<td>51</td>
</tr>
<tr>
<td>Configuration Time</td>
<td>51</td>
</tr>
<tr>
<td>Configuration Debugging</td>
<td>52</td>
</tr>
<tr>
<td><strong>Chapter 5: Known Issues and Limitations</strong></td>
<td></td>
</tr>
<tr>
<td>Known Issues</td>
<td>55</td>
</tr>
<tr>
<td>Known Limitations</td>
<td>56</td>
</tr>
<tr>
<td><strong>Appendix A: Additional Resources</strong></td>
<td></td>
</tr>
<tr>
<td>Xilinx Resources</td>
<td>57</td>
</tr>
<tr>
<td>Solution Centers</td>
<td>57</td>
</tr>
<tr>
<td>References</td>
<td>57</td>
</tr>
</tbody>
</table>
Overview

Partial Reconfiguration allows for the dynamic change of modules within an active design. This flow requires the implementation of multiple configurations which ultimately results in full bitstreams for each configuration, and partial bitstreams for each Reconfigurable Module.

The number of configurations required varies by the number of modules that need to be implemented. However, all configurations use the same top-level, or static, placement and routing results. These static results are exported from the initial configuration, and imported by all subsequent configurations using checkpoints.

This guide:

- Is intended for designers who want to create a partially reconfigurable FPGA design.
- Assumes familiarity with FPGA design software, particularly Xilinx® Vivado® Design Suite.
- Has been written specifically for Vivado Design Suite Release 2013.4. This release supports Partial Reconfiguration for Virtex®-7, Kintex®-7 and Zynq®-7000 AP SoC devices.
- Describes Partial Reconfiguration as implemented in the Vivado toolset.
Introduction to Partial Reconfiguration

FPGA technology provides the flexibility of on-site programming and re-programming without going through re-fabrication with a modified design. Partial Reconfiguration (PR) takes this flexibility one step further, allowing the modification of an operating FPGA design by loading a partial configuration file, usually a partial BIT file. After a full BIT file configures the FPGA, partial BIT files can be downloaded to modify reconfigurable regions in the FPGA without compromising the integrity of the applications running on those parts of the device that are not being reconfigured.

Figure 1-1 illustrates the premise behind Partial Reconfiguration.

As shown, the function implemented in Reconfig Block A is modified by downloading one of several partial BIT files, A1.bit, A2.bit, A3.bit, or A4.bit. The logic in the FPGA design is divided into two different types, reconfigurable logic and static logic. The gray area of the FPGA block represents static logic and the block portion labeled Reconfig Block "A" represents reconfigurable logic. The static logic remains functioning and is unaffected by the loading of a partial BIT file. The reconfigurable logic is replaced by the contents of the partial BIT file.

There are many reasons why the ability to time multiplex hardware dynamically on a single FPGA device is advantageous. These include:

- Reducing the size of the FPGA device required to implement a given function, with consequent reductions in cost and power consumption
- Providing flexibility in the choices of algorithms or protocols available to an application
- Enabling new techniques in design security
- Improving FPGA fault tolerance
- Accelerating configurable computing

In addition to reducing size, weight, power and cost, Partial Reconfiguration enables new types of FPGA designs that are impossible to implement without it.
Terminology

The following terminology is specific to the Partial Reconfiguration feature and is used throughout this document.

Bottom-Up Synthesis

Bottom-Up Synthesis is synthesis of the design by modules, whether in one project or multiple projects. Bottom-Up Synthesis requires that a separate netlist is written for each Partition, and no optimizations are done across these boundaries, ensuring that each portion of the design is synthesized independently. Top-level logic must be synthesized with black boxes for Partitions.

Configuration

A Configuration is a complete design that has one Reconfigurable Module for each Reconfigurable Partition. There may be many Configurations in a Partial Reconfiguration FPGA project. Each Configuration generates one full BIT file as well as one partial BIT file for each Reconfigurable Module.

Configuration Frame

Configuration frames are the smallest addressable segments of the FPGA configuration memory space. Reconfigurable frames are built from discrete numbers of these lowest-level elements. In a 7 series device, the base reconfigurable frames are one element (CLB, BRAM, DSP) wide by one clock region high.

Internal Configuration Access Port (ICAP)

The Internal Configuration Access Port (ICAP) is essentially an internal version of the SelectMAP interface. For more information, see the 7 Series FPGAs Configuration User Guide (UG470) [Ref 1].

Partial Reconfiguration (PR)

Partial Reconfiguration is modifying a subset of logic in an operating FPGA design by downloading a partial bitstream.

Partition

A Partition is a logical section of the design, defined by the user at a hierarchical boundary, to be considered for design reuse. A Partition is either implemented as new or preserved from a previous implementation. A Partition that is preserved maintains not only identical functionality but also identical implementation.
**Partition Pin**

Partition pins are the logical and physical connection between static logic and reconfigurable logic. Partition pins are automatically created for all Reconfigurable Partition ports.

**Processor Configuration Access Port (PCAP)**

The Processor Configuration Access Port (PCAP) is an alternative to the Internal Configuration Access Port (ICAP) for configuring a Zynq-7000 device. For more information, see the *Zynq-7000 All Programmable Technical Reference Manual* (UG585) [Ref 2].

**Reconfigurable Frame**

Reconfigurable frames (in all references other than "configuration frames" in this guide) represent the smallest reconfigurable region within an FPGA device. Bitstream sizes of reconfigurable frames vary depending on the types of logic contained within the frame.

**Reconfigurable Logic**

Reconfigurable Logic is any logical element that is part of a Reconfigurable Module. These logical elements are modified when a partial BIT file is loaded. Many types of logical components may be reconfigured such as LUTs, flip-flops, BRAM, and DSP blocks.

**Reconfigurable Module (RM)**

A Reconfigurable Module (RM) is the netlist or HDL description that is implemented within a Reconfigurable Partition. Multiple Reconfigurable Modules will exist for a Reconfigurable Partition.

**Reconfigurable Partition (RP)**

Reconfigurable Partition (RP) is an attribute set on an instantiation that defines the instance as reconfigurable. The Reconfigurable Partition is the level of hierarchy within which different Reconfigurable Modules will be implemented. Tcl commands such as `opt_design`, `place_design` and `route_design` detect the HD.RECONFIGURABLE property on the instance and process it correctly.

**Static Logic**

Static Logic is any logical element that is not part of a Reconfigurable Partition. The logical element is never partially reconfigured and is always active when Reconfigurable Partitions are being reconfigured. Static Logic is also known as Top-level Logic.
**Static Design**

The Static Design is the part of the design that will not change during partial reconfiguration. The static design includes the top level and all modules not defined as reconfigurable. The Static Design is built with Static Logic and Static Routing.

---

**Design Considerations**

Partial Reconfiguration (PR) is an expert flow within the Vivado Design Suite. Prospective customers must understand the following requirements and expectations before embarking on a PR project.

**Design Requirements and Guidelines**

- Partial Reconfiguration requires the use of Vivado 2013.3 or newer.
  - Partial Reconfiguration is supported in the ISE Design Suite as well. Please consult the *Partial Reconfiguration User Guide* (UG702) [Ref 3] for more information.
- Device support: Kintex-7, Virtex-7 T and XT (including 7V2000T and 7VX1140T), and four Zynq AP SoC devices: 7Z045, 7Z030, 7Z020 and 7Z010.
  - Artix-7, Virtex-7 HT and the remaining Zynq devices are not yet supported. These devices will become available in future software releases.
- PR is supported via Tcl or command line only; there is no project support at this time.
- Floorplanning is required to define reconfigurable regions, per element type.
  - For greatest efficiency, and to use the RESET_AFTER_RECONFIG feature, vertically align to frame/clock region boundaries.
  - Horizontal alignment rules also apply. See Create a Floorplan for the Reconfigurable Region in Chapter 2 for more information.
- Bottom-up synthesis (to create multiple netlist files) and management of Reconfigurable Module netlist files is the responsibility of the user.
  - Any synthesis tool may be used. Disable I/O insertion to create Reconfigurable Module netlists.
  - Vivado Synthesis uses the out-of-context Module Analysis flow for Reconfigurable Module synthesis.
- Standard timing constraints are supported, and additional timing budgeting capabilities are available if needed.
- A unique set of Design Rule Checks (DRCs) has been established to guide users on a successful path to design completion.
• A PR design must consider the initiation of Partial Reconfiguration as well as the delivery of partial BIT files, either within the FPGA or as part of the system design.

• A Reconfigurable Partition must contain a super set of all pins to be used by the varying Reconfigurable Modules implemented for the partition. It is expected that this will lead to unused inputs or outputs for some modules, and is designed into the flexibility of the PR solution. The unused inputs will be left dangling inside of the module; drive outputs to a constant if this is an issue for your design.

  - In the case of a black box RM (no logic) after update_design –black_box has been issued, all partition pin outputs will be undriven. If this black box module will be used to create a bitstream, it is recommended that decoupling logic for this RP remain active while the black box is loaded in the device.

**Design Performance**

Performance metrics will vary from design to design, and the best results will be seen when if you follow the Hierarchical Design techniques documented in the Hierarchical Design Methodology Guide (UG748) [Ref 4], and in Repeatable Results with Design Preservation (WP362) [Ref 5]. These documents were created for the ISE Design Suite, but the methodologies contained therein still apply for the Vivado Design Suite.

However, the additional restrictions that are required for silicon isolation are expected to have an impact on most designs. The application of Partial Reconfiguration rules, such as routing containment, exclusive placement, and no optimization across reconfigurable module boundaries, will mean that the overall density and performance will be lower for a PR design than for the equivalent flat design. The overall design performance for PR designs will vary from design to design based on factors such as the number of reconfigurable partitions, the number of interface pins to these partitions, and the size and shape of Pblocks.

**Design Criteria**

• Some component types can be reconfigured and some cannot.

  - Reconfigurable resources include CLB, BRAM, and DSP component types as well as routing resources.

  - Clocks and Clock modifying logic cannot be reconfigured, and therefore must reside in the static region.

    - Includes BUFG, BUFR, MMCM, PLL, and similar components
- The following components cannot be reconfigured, and therefore must reside in the static region:
  - I/O and I/O related components (ISERDES, OSERDES, IDELAYCTRL, etc.)
  - Serial transceivers (MGTs) and related components
  - Individual architecture feature components (such as BSCAN, STARTUP, XADC, etc.) must remain in the static region of the design
- Global clocking resources to Reconfigurable Partitions are limited, depending on the device and on the clock regions occupied by these Reconfigurable Partitions.
- IP restrictions may occur due to components used to implement the IP. Examples include:
  - Vivado Debug Hub (BSCAN and BUFG)
  - IP modules with embedded global buffers or I/O
  - MIG controller (MMCM)
- Reconfigurable Modules must be initialized to ensure a predictable starting condition after reconfiguration. You can do this manually with a local reset, or via dedicated GSR events by selecting the RESET_AFTER_RECONFIG feature.
- Decoupling logic is highly recommended to disconnect the reconfigurable region from the static portion of the design during the act of Partial Reconfiguration.
  - Clock and other inputs to Reconfigurable Modules can be decoupled to prevent spurious writes to memories during reconfiguration. This should be considered if RESET_AFTER_RECONFIG is not used.
- A reconfigurable partition must be floorplanned, so the module must be a block that can be contained by a Pblock and meet timing. If the module is complete, it is recommended to run this design through a non-PR flow to get an initial evaluation of placement, routing, and timing results. If the design has issues in a non-PR flow, these should be resolved before moving on to the PR flow.
- Each module pin on an RP will have a partition pin. This is a routing point that connects static logic to the RP. If a design has too many partition pins for the number of available routing resources, routing congestion can occur. Consider the number of external pins on the RP, and pick a module that has a minimum required set of pins.
- Virtex-7 SSI devices (7V2000T, 7VX1140T) have two fundamental requirements. These requirements, which will extend to Virtex-7 HT devices when support becomes available, are:
  - Reconfigurable regions must be fully contained within a single SLR. This ensures that the global reset events are properly synchronized across all elements in the Reconfigurable Module, and that all Super Long Lines (SLL) are contained within the static portion of the design. SLL are not partially reconfigurable.
If the ICAP is used for partial bitstream delivery, it must be one located on the Master SLR, which is SLR1 for these devices. Apply a location constraint on the ICAP to the ICAP_X0Y2 or ICAP_X0Y3 locations only. The bitstream format is such that the standard daisy chain through the four SLRs is maintained. Do not use an ICAP on any of the other SLRs, even if the reconfigurable region is located there.

- Dedicated encryption support for partial bitstreams will be available natively for 7 series devices in a future Vivado release.
- 7 series devices can utilize a per-frame CRC checking mechanism, enabled via `write_bitstream`, to ensure each frame is valid before loading.

**Note:** This feature will be implemented in a future Vivado release.

Partial Reconfiguration is a powerful capability within Xilinx FPGAs, and understanding the capabilities of the silicon and software is instrumental to success with this technology. While trade-offs must be recognized and considered during the development process, the overall result will be a more flexible implementation of your FPGA design.
Vivado Software Flow

Software Flow Overview

The Vivado® Partial Reconfiguration design flow is similar to a standard design flow, with some notable departures. The implementation software automatically manages the low-level details to meet silicon requirements. The user must provide guidance to define the design structure and floorplan. The steps for processing a PR design can be summarized as follows:

1. Synthesize the static and Reconfigurable Modules separately.
2. Create physical constraints (Pblocks) to define the reconfigurable regions.
3. Set the HD.RECONFIGURABLE property on each Reconfigurable Partition.
4. Implement a complete design (static and one Reconfigurable Module per Reconfigurable Partition) in context.
5. Save a design checkpoint for the full routed design.
6. Remove Reconfigurable Modules from this design and save a static-only design checkpoint.
7. Lock the static placement and routing.
8. Add new Reconfigurable Modules to the static design and implement this new configuration.
9. Repeat Step 8 until all Reconfigurable Modules are implemented.
10. Run a verification utility (pr_verify) on all configurations.
11. Create bitstreams for each configuration.
Partial Reconfiguration Commands

The PR flows are currently only supported through the non-project batch/Tcl interface (no project based commands). Example scripts are provided in the *Vivado Design Suite Tutorial: Partial Reconfiguration* (UG947) [Ref 6], along with step by step instructions for setting up the flows. See that Tutorial for more information.

The following sections describe a few specialized commands and options needed for the PR flows. Examples of how to use these commands to run a PR flow are given. For more information on individual commands, see the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 7].

Synthesis

Synthesizing a partially reconfigurable design does not require any special commands, but does require bottom-up synthesis. There are currently no unsupported commands for synthesis, optimization, or implementation.

These synthesis tools are supported:

- XST
- Synplify
- Vivado Synthesis

**IMPORTANT:** Bottom-up synthesis refers to a synthesis flow in which each module has its own synthesis project. This generally involves turning off automatic I/O buffer insertion for the lower level modules.

This document only covers the Vivado Synthesis flow. For information on the other flows, refer to the *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices* (UG687) [Ref 8], or the Synopsys Synplify documentation.

Synthesizing the Top Level

You must have a top-level netlist with a black box for each Reconfigurable Module (RM). This requires the top-level synthesis to have module/entity declarations for the partitioned instances, but no logic – the module is empty.

The top-level synthesis will infer or instantiate I/O buffers on all top level ports; I/O logic inside of a Reconfigurable Module is not supported. For more information on controlling buffer insertion, refer to the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 9].

```
synth_design -flatten_hierarchy rebuilt -top <top_module_name> -part <part>
```
**Synthesizing Reconfigurable Modules**

Because each Reconfigurable Module must be instantiated in the same black box in the static design, the different versions must have identical interfaces. The name of the block must be the same in each instance, and all the properties of the interfaces (names, widths, direction) must also be identical. Each configuration of the design will be assembled like a flat design.

To synthesize a Reconfigurable Module, all buffer insertion must be turned off. This can be done in Vivado Synthesis using the `synth_design` command in conjunction with the `-mode out_of_context` switch:

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

**Table 2-1: synth_design Options**

<table>
<thead>
<tr>
<th>Command Option</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mode out_of_context</code></td>
<td>Prevents I/O insertion for synthesis and downstream tools.</td>
</tr>
<tr>
<td></td>
<td>The <code>out_of_context</code> mode will be saved in the checkpoint if <code>write_checkpoint</code> is issued.</td>
</tr>
<tr>
<td><code>-flatten_hierarchy rebuilt</code></td>
<td>There are several values allowed for <code>-flatten_hierarchy</code>, but <code>rebuilt</code> is the recommended setting for PR flows.</td>
</tr>
<tr>
<td><code>-top</code></td>
<td>This is the module/entity name of the module being synthesized.</td>
</tr>
<tr>
<td><code>-part</code></td>
<td>This is the Xilinx® part being targeted (for example, xc7k325tffg900-3)</td>
</tr>
</tbody>
</table>

The `synth_design` command synthesizes the design and stores the results in memory. In order to write the results out to a file, use:

```
write_checkpoint <file_name>.dcp
```

It is recommended to close the design in memory after synthesis, and run implementation separately from synthesis.

**Reading Design Modules**

If there is currently no design in memory, then a design must be loaded. This can be done in a variety of ways, for either the static design or for Reconfigurable Modules. After configurations have been implemented, checkpoints will be exclusively used to read in placed and routed module databases.
Method 1: Read Netlist Design

This approach should be used when modules have been synthesized by tools other than Vivado Synthesis.

```plaintext
read_edif <top>.edf/edn/ngc
read_edif <rp1_a>.edf/edn/ngc
read_edif <rp2_a>.edf/edn/ngc
link_design -top <top_module_name> -part <part>
```

Table 2-2: link_design Options

<table>
<thead>
<tr>
<th>Command Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-part</td>
<td>This is the Xilinx part being targeted (for example, xc7k325tffg900-3)</td>
</tr>
<tr>
<td>-top</td>
<td>This is the module/entity name of the module being implemented. This switch can be omitted if set_property top &lt;top_module_name&gt; [current_fileset] is issued prior to link_design.</td>
</tr>
</tbody>
</table>

Method 2: Open/Read Checkpoint

If the static (top-level) design has synthesis or implementation results stored as a checkpoint, then it can be loaded using the `open_checkpoint` command. This command reads in the static design checkpoint and opens it in active memory.

```plaintext
open_checkpoint <file>
```

If the checkpoint is for a reconfigurable module (i.e., not for static), then the instance name must be specified using `read_checkpoint -cell`. If the checkpoint is a post-implementation checkpoint, then the additional `-strict` option must be used as well. This option can also be used with a post-synthesis checkpoint to ensure exact port matching has been achieved. To read in a Reconfigurable Module's checkpoint, the top-level design must already be opened, and must have a black box for the specified cell. Then the following command can be specified:

```plaintext
read_checkpoint -cell <cellname > <file> [-strict]
```

Table 2-3: read_checkpoint Switches

<table>
<thead>
<tr>
<th>Switch Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-cell</td>
<td>Used to specify the full hierarchical name of the Reconfigurable Module.</td>
</tr>
<tr>
<td>-strict</td>
<td>Requires exact ports match for replacing cell, and checks that part, package, and speed grade values are identical. Should be used when restoring implementation data.</td>
</tr>
<tr>
<td>&lt;file&gt;</td>
<td>Specifies the full or relative path to the checkpoint (DCP) to be read in.</td>
</tr>
</tbody>
</table>
Method 3: Open Checkpoint/Update Design

This is useful when the synthesis results are in the form of a netlist (edf/edn/ngc), but static has already been implemented. The following example shows the commands for the second configuration in which this is true.

```
open_checkpoint <top>.dcp
lock_design -level routing
update_design -cells <rp1> -from_file <rp1_b>.{edf/edn/ngc}
update_design -cells <rp2> -from_file <rp2_b>.{edf/edn/ngc}
```

Adding Reconfigurable Modules with Multiple Netlists

If a Reconfigurable Module has sub-module netlists, it can be difficult for the Vivado tools to process the sub-module netlists. This is because in the PR flow the RM netlists are added to a design that is already open in memory. This means the `update_design -cells` command must be used, which requires the cell name for every EDIF file, which can be troublesome to get.

There are two ways to make loading RM sub-module netlists easier in the Vivado Design Suite.

Method 1: Create a Single RM Checkpoint (DCP)

Create an RM checkpoint (DCP) that includes all netlists. In this case all of the EDIF files can be added using `add_files`, and `link_design` can be used to resolve the EDIF files to their respective cells. Here is an example of the commands used in this process:

```
add_files [list rm.edf ip_1.edf … ip_n.edf]
# Run if RM XDC exists
add_files rm.xdc
link_design -top <rm_module> -part <part>
write_checkpoint rm_v#.dcp
close_project
```

Then this newly-created RM checkpoint can be used in the PR flow. In the commands below, the single `read_checkpoint -cell` command replaces what could be many `update_design -cell` commands.

```
add_files static.dcp
link_design -top <top> part <part>
lock_design -level routing
read_checkpoint -cell <rm_inst> rm_v#.dcp
```
Method 2: Place the Sub-Module Netlists in the Same Directory as the RM’s Top-Level Netlist

When the top-level RM netlist is read into the PR design using `update_design -cell`, make sure that all sub-module netlists are in the same directory as the RM’s top-level netlist. In this case the lower level netlists do not need to be specified, but they will be picked up automatically by the `update_design -cells` command. This is less explicit than Method 1, but requires fewer steps. In this case the commands to load the RM netlist would look like the following.

```
add_files static.dcp
link_design -top <top> part <part>
lock_design -level routing
update_design -cells <rm_inst> -from_file rm_v#.edf
```

In the last (`update_design`) command above, the lower level netlists will get picked up automatically if they are in the same directory as `rm_v#.edf`.

Implementation

Since the PR flow allows for various configurations in hardware, multiple implementation runs are required. Each implementation of a PR design is referred to as a configuration. Each module of the design (static or Reconfigurable Module) can be implemented or imported (if previously implemented). Implementation results for the static design must be consistent for each configuration, so it will be implemented in one configuration, and then imported in subsequent configurations. Additional configurations can be constructed by importing static, and implementing/importing each Reconfigurable Module.

There are no restrictions to the support of implementation commands or options for PR, but certain optimizations and sub-routines will not be done if they oppose the fundamental requirements of partial reconfiguration. The commands that may be run once the logical design is loaded (via `link_design` or `open_checkpoint`) are listed below.

```
# Run if all constraints are not already loaded
read_xdc

# Optional command
opt_design

place_design

# Optional command
phys_opt_design

route_design
```
Preserving Implementation Data

In the PR flow, it is a requirement to lock down the placement and routing results of the static logic from the first configuration for all subsequent configurations. The static implementation of the first configuration must be saved as a checkpoint. When the checkpoint is read for subsequent configurations, the placement and routing must be locked, to ensure that the static design remains completely identical from configuration to configuration. To lock the placement and routing of an imported checkpoint (static or reconfigurable), the `lock_design` command is used.

```
lock_design -level routing [cell_name]
```

When locking down the static logic with the above command, the optional `[cell_name]` can be omitted.

```
lock_design -level routing
```

To lock the results of an imported RM, the full hierarchical name should be specified within the post-implementation checkpoint:

```
lock_design -level routing u0_RM_instance
```

For Partial Reconfiguration, the only supported preservation level is `routing`. Other preservation levels are available for this command, but they must only be used for other Hierarchical Design flows.

---

Partial Reconfiguration Constraints and Properties

There are a few properties and constraints unique to the Partial Reconfiguration flow. These initiate PR-specific implementation processing and apply specific characteristics in the partial bitstreams. The four areas for constraints and properties for partial reconfiguration are:

- Defining a module as reconfigurable - required
- Creating a floorplan for the reconfigurable region - required
- Applying reset after reconfiguration - optional, but highly recommended
- Turn on visualization scripts - optional

Define a Module as Reconfigurable

In order to implement a PR design, it is required to specify each Reconfigurable Module as such. To do this you must set a property on the top level of each hierarchical cell that is going to be reconfigurable. For example, take a design where one Reconfigurable Partition named "inst_count" exists, and it has two Reconfigurable Modules, "count_up" and
"count_down". The following command must be issued prior to implementation of the first configuration.

```bash
set_property HD.RECONFIGURABLE TRUE [get_cells inst_count]
```

This will initiate the Partial Reconfiguration features in the software that are required to successfully implement a PR design. The HD.RECONFIGURABLE property implies a number of underlying constraints and tasks:

- Sets DONT_TOUCH on the specified cell and its interface nets. This prevents optimization across the boundary of the module.
- Sets EXCLUDE_PLACEMENT on the cell's Pblock. This prevents static logic from being placed in the reconfigurable region.
- Sets CONTAIN_ROUTING on the cell's Pblock. This keeps all the routing for the Reconfigurable Module within the bounding box.
- Enables special code for DRCs, clock routing, etc.

**Create a Floorplan for the Reconfigurable Region**

Each Reconfigurable Partition is required to have a Pblock to define the physical resources available for the Reconfigurable Module. Because this Pblock will be set on a Reconfigurable Partition, these restrictions and requirements apply:

- The Pblock must contain only CLB/SLICE, DSP and BRAM sites. The region may overlap other site types, but these other sites must not be included in the `resize_pblock` commands.
- Multiple Pblock rectangles for each component type may be used to create the Reconfigurable Partition region, but for the greatest routability, they should be contiguous. Gaps to account for non-reconfigurable resources are permitted.
- If using the RESET_AFTER_RECONFIG property, the Pblock height must align to clock region boundaries. See [Apply Reset After Reconfiguration](#) for more detail.
- The width and composition of the Pblock must not split interconnect columns. See [Reconfigurable Partition Pblock Sizes and Shapes in Chapter 3](#) for more detail.
- The Pblock must not overlap any other Pblock in the design.
- Nesting of Reconfigurable Partitions (a Reconfigurable Partition within another Reconfigurable Partition) is currently not supported.
Partial Reconfiguration Constraints and Properties

Table 2-4: Pblock Commands and Properties

<table>
<thead>
<tr>
<th>Command/Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_pblock</td>
<td>Command used to create the initial Pblock for each Reconfigurable Partition instance.</td>
</tr>
<tr>
<td>add_cells_to_pblock</td>
<td>Command used to specify the instances that will belong to the Pblock. This is typically a level of hierarchy as defined by the bottom-up synthesis processing.</td>
</tr>
<tr>
<td>resize_pblock</td>
<td>Command used to define the site types (SLICE, RAMB36, etc.) and site locations that will be owned by the Pblock.</td>
</tr>
<tr>
<td>RESET_AFTER_RECONFIG</td>
<td>Pblock property used to control the use of the dedicated GSR event on the reconfigurable region. Use of this property is highly recommended and requires clock region alignment in the vertical direction.</td>
</tr>
<tr>
<td>CONTAIN_ROUTING</td>
<td>Pblock property used to control the routing to prevent usage of routing resources not owned by the Pblock. This property is mandatory for PR and is set to True automatically for Reconfigurable Partitions. Static routing is still allowed to use resources inside of the Pblock.</td>
</tr>
<tr>
<td>EXCLUDE_PLACEMENT</td>
<td>Pblock Property used to prevent the placement of any logic, not belonging to the Pblock, inside the defined Pblock RANGE. This property is mandatory for PR and set to true automatically for Reconfigurable Partitions. Static logic can be placed inside of the Reconfigurable Partition with a specific LOC property if RESET_AFTER_RECONFIG is not used.</td>
</tr>
</tbody>
</table>

Here is an example of a set of constraints for a Reconfigurable Partition.

```plaintext
#define a new pblock
create_pblock pblock_count

#add a hierarchical module to the pblock
add_cells_to_pblock [get_pblocks pblock_count] [get_cells [list inst_count]]

#define the size and components within the pblock
resize_pblock [get_pblocks pblock_count] -add {SLICE_X136Y50:SLICE_X145Y99}
resize_pblock [get_pblocks pblock_count] -add {RAMB18_X6Y20:RAMB18_X6Y39}
resize_pblock [get_pblocks pblock_count] -add {RAMB36_X6Y10:RAMB36_X6Y19}
```
Floorplan in the Vivado IDE

Even though Project Mode support is not available, the Vivado IDE can be used for planning and visualization tasks. The best example of this is the use of the Device view to create and modify Pblock constraints for floorplanning. First, open the synthesized static design and the largest of each Reconfigurable Module. Here are the commands, using the tutorial design (found in the *Vivado Design Suite Tutorial: Partial Reconfiguration* (UG947) [Ref 6]) as an example:

```plaintext
open_checkpoint synth/Static/top_synth.dcp
read_checkpoint -cell [get_cells inst_count] synth/count_up/count synth.dcp
read_checkpoint -cell [get_cells inst_shift] synth/shift_right/shift synth.dcp
set_property HD.RECONFIGURABLE true [get_cells inst_count]
set_property HD.RECONFIGURABLE true [get_cells inst_shift]
```

At this point, a full configuration has been loaded into memory, and the Reconfigurable Partitions have been defined. To create Pblock constraints for the Reconfigurable Partitions, right-click on an instance in the Netlist window (in this case, inst_count or inst_shift) and select **Draw Pblock**. Create a rectangle in the Device view to select resources for this Reconfigurable Partition.

With this Pblock selected, note that the Pblock Properties pane will show the number of available and required resources. The number required is based on the currently loaded Reconfigurable Module, so keep in mind that other modules may have different requirements. If additional rectangles are required to build the appropriate shape (an "L", for example), right-click the Pblock in the Device view and select **Add Pblock Rectangle**.

Design Rule Checks (DRCs) can be issued to validate the floorplan and other design considerations for the in-memory configuration. To run, select **Tools > Report > Report DRC** and ensure the Partial Reconfiguration checks are present (see Figure 2-1). Note that if HD.RECONFIGURABLE has not been set on a Pblock, only a single DRC will be available, instead of the full complement shown below.
This set of DRCs can be run from the Tcl Console or within a script, by using the `report_drc` command. To limit the checks to the ones shown here for Partial Reconfiguration, use this syntax:

```
report_drc -checks [get_drc_checks HDPR*]
```
To extend the DRCs to those checked during specific phases of design processing the `-ruledeck` option can be used. For example, the following command can be issued on a placed and routed design:

```
report_drc -ruledeck bitstream_checks
```

To save these floorplanning constraints, enter the following command in the Tcl Console:

```
write_xdc top_fplan.xdc
```

The Pblock constraints stored in this constraints file can be used directly or can be copied to another top-level design constraints file. This XDC file will contain all the constraints in the current design in memory, not just the constraints recently added.

**CAUTION!** Do NOT save the overall design from the Vivado IDE using **File > Save Checkpoint** or the equivalent icon. If you save the currently loaded design in this way, you will overwrite your synthesized static design checkpoint with a new version that includes Reconfigurable Modules and additional constraints.

### Partition Pins

Interface points called partition pins are automatically created within the Pblock ranges defined for the Reconfigurable Partition. These virtual I/O are established within interconnect tiles as the anchor points that remain consistent from one module to the next. No physical resources such as LUTs or flip-flops are required to establish these anchor points, and no additional delay is incurred at these points.

The placer chooses locations based on source and loads and timing requirements, but these locations can be driven by the user. The following constraints may be applied to influence partition pin placement.

**Table 2-5: Context Properties**

<table>
<thead>
<tr>
<th>Command/Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HD.PARTPIN_LOCS</td>
<td>Used to define a specific interconnect tile (INT) for the specified port to be routed. Overrides an HD.PARTPIN_RANGE value. Affects placement and routing of logic on both sides of the Reconfigurable Partition boundary. Do not use this property on clock ports, as this will assume local routing for the clock. Do not use this property on dedicated connections.</td>
</tr>
<tr>
<td>HD.PARTPIN_RANGE</td>
<td>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 this property on dedicated connections.</td>
</tr>
</tbody>
</table>
Context Property Examples:

- `set_property HD.PARTPIN_LOCS INT_R_X4Y153 [get_ports <port_name>]`
- `set_property HD.PARTPIN_RANGE SLICE_X4Y153:SLICE_X5Y157 [get_ports <port_name>]`

Instance names for interconnect tile sites can be seen in the Device View with the Routing Resources enabled.

### Apply Reset After Reconfiguration

With the Reset After Reconfiguration feature, the reconfiguring region is held in a steady state during partial reconfiguration, and then all logic in the new Reconfigurable Module is initialized to its starting values. Static routes may still freely pass unaffected through the region, and static logic (and all other PR regions) elsewhere in the device will continue to operate normally during Partial Reconfiguration. Partial Reconfiguration with this feature will behave just like the initial configuration of the FPGA, with synchronous elements being released in a known, initialized state.

This is the `RESET_AFTER_RECONFIG` property syntax:

```
set_property RESET_AFTER_RECONFIG true [get_pblocks <reconfig_pblock_name>]
```

In order to apply the Reset After Reconfiguration methodology, Pblock constraints must align to reconfigurable frames. Because the GSR will affect every synchronous element within the region, exclusive use of reconfiguration frames is required; static logic is not permitted within these reconfigurable frames (static routing is permitted). Pblocks must align vertically to clock regions, since that matches the base region for a reconfigurable frame.

In Figure 2-2, the Pblock on the left (pblock_shift) is permitted to use the `RESET_AFTER_RECONFIG` feature because the top and bottom of the Pblock align to the height of clock region X1Y3. The Pblock on the right (pblock_count) cannot use `RESET_AFTER_RECONFIG` because the top is not aligned, and any static logic placed between it and the clock region boundary above it would be affected by GSR after that module was partially reconfigured.
An alternative approach would be to forego this property and apply a local reset to any reconfigured logic that requires initialization to function properly. This approach does not require vertical alignment to clock region boundaries. Without GSR or a local reset, the initial starting value of a synchronous element within a reconfigured module cannot be guaranteed.

**Turn On Visualization Scripts**

The configuration tiles that are part of the partial bitstreams can be visualized within the Device View in the Vivado IDE. These are identified via scripts that are created during implementation. To turn on script creation, set this parameter before starting implementation:

```
set_param HD.VISUAL true
```

This will generate multiple scripts placed in the `hd_visual` directory, which is created in the directory where the run script is launched. To use these scripts, read a routed design checkpoint into the Vivado IDE, then source one of the scripts. These design-specific scripts will highlight configuration tiles as defined by the user, show configuration frames used to

---

**Figure 2-2:** RESET_AFTER_RECONFIG Compatible (Left) and Incompatible (Right) Pblocks
Software Flow

This section describes the basic flow, and gives sample commands to execute this flow.

Synthesis

Each module (including Static) needs to be synthesized bottom-up so that a netlist/checkpoint exists for static and each Reconfigurable Module.

1. Synthesize the top level

   read_verilog top.v (and other HDL associated with the static design, including black box module definitions for Reconfigurable Modules)

   then:

   read_xdc top_synth.xdc
   synth_design -top top -part xc7k70tfbg676-2
   write_checkpoint top_synth.dcp

2. Synthesize a Reconfigurable Module

   read_verilog rp1_a.v
   synth_design -top rp1 -part xc7k70tfbg676-2 -mode out_of_context
   write_checkpoint rp1_a_synth.dcp

3. Repeat for each remaining Reconfigurable Module

   read_verilog rp1_b.v
   synth_design -top rp1 -part xc7k70tfbg676-2 -mode out_of_context
   write_checkpoint rp1_b_synth.dcp

create the partial bit file, or show sites excluded by the PR floorplan. Additional scripts are created for other flows, such as Module Analysis or Tandem Configuration, and are not used for PR.
Implementation

Create as many configurations as necessary to implement all Reconfigurable Modules at least once. The first configuration loads in synthesis results for top and the first Reconfigurable Module. You must then mark the module as being reconfigurable, then run implementation. Write out a checkpoint for the Reconfigurable Module so it can be reused later if desired. Finally, remove the Reconfigurable Module from the design (update_design -cell -black_box) and write out a checkpoint for the static design alone.

Configuration 1:

```
open_checkpoint top_synth.dcp
read_xdc top_impl.xdc
read_checkpoint -cell rp1 rp1_a_synth.dcp
set_property HD.RECONFIGURABLE true [get_cells rp1]
opt_design
place_design
route_design
write_checkpoint config1_routed.dcp
write_checkpoint -cell rp1 rp1_a_route_design.dcp
update_design -cell rp1 -black_box
write_checkpoint static_routed.dcp
```

For the second configuration, load the placed and routed checkpoint for static (if it was closed) which currently has a black box for the Reconfigurable Module, and lock it down. Then load in the synthesis results for the second Reconfigurable Module and implement the design. Finally write out an implementation checkpoint for the second version of the Reconfigurable Module.

Configuration 2:

```
open_checkpoint static_routed.dcp
lock_design -level routing
read_checkpoint -cell rp1 rp1_b_synth.dcp
opt_design
place_design
route_design
write_checkpoint config2_routed.dcp
write_checkpoint -cell rp1 rp1_b_route_design.dcp
```

**TIP:** Keep each configuration in a separate folder so that all intermediate checkpoints, log and report files, bit files, and other design outputs are kept unique.

If multiple Reconfigurable Partitions exist, then other configurations may be required. Additional configurations can also be created by importing previously implemented
Reconfigurable Modules to create full designs that will exist in hardware. This can be useful for creating full bitstreams with a desired combination for power-up, or for performing static timing analysis, power analysis, or simulation.

**IMPORTANT:** See Known Issues in Chapter 5 for a current issue with reuse of implemented Reconfigurable Module checkpoints.

### Reporting

Each step of the implementation flow will perform design rule checks (DRCs) unique to partial reconfiguration. Keep a close eye on the messages given by the implementation steps to ensure no critical warnings are issued. These messages will guide designers to optimize module interfaces, floorplans, and other key aspects of PR designs.

Reports that can be generated do not have PR-specific sections, but useful information can be extracted nonetheless. For example, utilization information can be obtained by using the `-pblocks` switch for the `report_utilization` command. This will show the used and available resources within a given reconfigurable module. Here is an example using the design from the Vivado Design Suite Tutorial: Partial Reconfiguration (UG947) [Ref 6]:

```
report_utilization -pblocks [get_pblocks pblock_count]
```

### Verifying Configurations

Once all configurations have been completely placed and routed, a final verification check can be done to validate consistency between these configurations using `pr_verify`. This command takes in multiple routed checkpoints (DCPs) as arguments, and outputs a log of any differences found in the static implementation and Partition Pin placement between them. Placement and routing within any RMs is ignored during the comparison.

When just two configurations are to be compared, list the two routed checkpoints as `<file1>` and `<file2>`. `pr_verify` will load both in memory and make the comparison. When more than two configurations are to be compared, provide a "master" configuration using the `-initial` switch, then list the remaining configurations by using the `-additional` switch, listing configurations in braces ({} and {}). The initial configuration is kept in memory and the remaining configurations will be compared against the initial one. Bitstreams should not be generated for any configurations if any pair of configurations do not pass the PR Verify check.

```
pr_verify [-full_check] [-file <arg>] [-initial <arg>] [-additional <arg>] [-quiet] [-verbose] [<file1>] [<file2>]
```
Here is a sample command line comparing two configurations:

```
pr_verify -full_check config1_routed.dcp config2_routed.dcp -file pr_verify_c1_c2.log
```

Here is a second example verifying three configurations:

```
pr_verify -full_check -initial config1.dcp -additional {config2.dcp config3.dcp} -file three_config.log
```

The scripts provided with the *Vivado Design Suite Tutorial: Partial Reconfiguration* (UG947) [Ref 6] have a Tcl Proc called "verify_configs" that automatically runs all existing configurations through `pr_verify`, and report if the DCPs are compatible or not.

### Bitstream Generation

Just like a flat flow, bitstreams are created with the `write_bitstream` command. For each design configuration, simply issue `write_bitstream` to create a full standard configuration file plus one partial bit file for each Reconfigurable Module within that configuration.

It is recommended to provide the configuration name and Reconfigurable Module names in the `-file` option specified with `write_bitstream`. Only the base bit file name can be modified, so it is important to record which Reconfigurable Modules were selected for each configuration.

Using the design above, here is an example of reading routed checkpoints (configurations) and creating bitstreams for all implemented Reconfigurable Modules.

```
read_checkpoint config1_routed.dcp
write_bitstream config1
```
This command creates a full design bitstream called `config1.bit`. This bitstream should be used to program the device from power-up and includes the functionality of any Reconfigurable Modules contained within. It will also create partial bit files `config1_pblock_rpl_partial.bit` and `config1_pblock_rpl2_partial.bit` that can be used to reconfigure these modules while the FPGA continues to operate. Repeat these steps for each configuration.

If a power-on configuration of the static design only is desired, run `write_bitstream` on the checkpoint that has empty Reconfigurable Partitions (after `-update_design -black_box` ran). This "black box configuration" can be compressed to reduce the bit file size and configuration time. The outputs of the RPs will be undriven, so the design should be structured to power up with the decoupling logic enabled.

**TIP:** Rename each partial bit file to match the Reconfigurable Module instance from which it was built to uniquely identify these modules. The current solution names partial bit files only on the configuration base name and Pblock name: `<base_name>_<pblock_name>_partial.bit`

Bitstream compression and other advanced features can be used.

**CAUTION!** Do not run `write_bitstream` directly on Reconfigurable Module checkpoints; only use full design checkpoints. Reconfigurable module checkpoints, while they are placed and routed submodules, have no information regarding the top level design implementation, and therefore would create unsuitable partial bit files.

---

**Tcl Scripts**

Scripts are provided to run this flow in the *Vivado Design Suite Tutorial: Partial Reconfiguration* (UG947) [Ref 6]. The details of these sample scripts are documented in the tutorial itself and in the readme.txt contained in the sample design archive.
Chapter 3

Design Considerations and Guidelines

Introduction

This chapter explains design requirements that are unique to Partial Reconfiguration, and covers specific PR features within the Xilinx® FPGA design software tools.

To take advantage of the Partial Reconfiguration capability of Xilinx FPGA devices, you must analyze the design specification thoroughly, and consider the requirements, characteristics, and limitations associated with PR designs. This simplifies both the design and debug processes, and avoids potential future risks of malfunction in the design.

Design Hierarchy

Good hierarchical design practices resolve many complexities and difficulties when implementing a Partially Reconfigurable FPGA design. A clear design instance hierarchy simplifies physical and timing constraints. Registering signals at the boundary between static and reconfigurable logic eases timing closure. Grouping logic that is packed together in the same hierarchical level is necessary.

These are all well known design practices that are often not followed in general FPGA designs. Following these design rules is not strictly required in a partially reconfigurable design, but the potential negative effects of not following them are more pronounced. The benefits of Partial Reconfiguration are great, but the extra complexity in design could be more challenging to debug, especially in hardware.

For additional information about design hierarchy, see:

- Hierarchical Design Methodology Guide (UG748) [Ref 4]
- Repeatable Results with Design Preservation (WP362) [Ref 5]
Design Elements Inside Reconfigurable Modules

Not all logic is permitted to be actively reconfigured. Global logic and clocking resources must be placed in the static region to not only remain operational during reconfiguration, but to benefit from the initialization sequence that occurs at the end of a full device configuration.

Logic that can be placed in a Reconfigurable Module includes:

- All logic components that are mapped to a CLB slice in the FPGA. This includes LUTs (look-up tables), FFs (flip-flops), SRLs (shift registers), RAMs, and ROMs.
- Block RAM (BRAM) and FIFO:
  - RAMB18E1, RAMB36E1, BRAM_SDP_MACRO, BRAM_SINGLE_MACRO, BRAM_TDP_MACRO
  - FIFO18E1, FIFO36E1, FIFO_DUALCLOCK_MACRO, FIFO_SYNC_MACRO
  
  Note: The IN_FIFO and OUT_FIFO design elements cannot be placed in an RM. These design elements must remain in static logic.
- DSP blocks: DSP48E1
- PCIe (PCI Express) - Entered using PCIe IP

All other logic must remain in static logic, and must not be placed in an RM, including:

- Clocks and Clock Modifying Logic - Includes BUFG, BUFR, MMCM, PLL, and similar components
- I/O and I/O related components (ISERDES, OSERDES, IDelayCTRL, etc.)
- Serial transceivers (MGTs) and related components
- Individual architecture feature components (such as BSCAN, STARTUP, XADC, etc.)

Dynamic Reconfiguration Using the DRP

Logic that must remain in the static region, and therefore is not available for Partial Reconfiguration, can still be reconfigured dynamically through the Dynamic Reconfiguration Port (DRP). The DRP can be used to configure logic elements such as MMCMs, PLLs, and serial transceivers (MGTs).

Information about the DRP and dynamic reconfiguration, including how to use the DRP for specific design resources, can be found in these documents:

- 7 Series FPGAs Configuration User Guide (UG470) [Ref 1]
- 7 Series FPGAs GTX/GTH Transceivers User Guide (UG476) [Ref 10]
- MMCM and PLL Dynamic Reconfiguration (7 Series) (XAPP888) [Ref 11]
Packing Logic

Any logic that must be packed together must be placed in the same group, whether it is static or reconfigurable. For example, if a LUT and a flip-flop are expected to be placed within the same slice, they must be within the same partition. Partition boundaries are barriers to optimization.

Design Instance Hierarchy

The simplest method is to instantiate the Reconfigurable Partitions in the top-level module, but this is not required since a Reconfigurable Partition may be located in any level of hierarchy. Each Reconfigurable Partition must correspond to exactly one instance – an RP may not have more than one top. The instantiation has multiple modules with which it is associated.

Reconfigurable Partition Interfaces

One of the fundamental requirements of a partially reconfigurable design is consistency between Reconfigurable Modules. As one module is swapped for another, the connections between the static design and the Reconfigurable Module must be identical, both logically and physically. In order to achieve this consistency, optimizations across the partition boundary or of the boundary itself are prohibited.

For optimal efficiency, all ports of a Reconfigurable Partition should be actively used on the static design side. For example, if static drivers of the Reconfigurable Partition are driven by constants (0 or 1), they will be implemented via the creation of a LUT instance and local tie-off to a constant driver and cannot be trimmed away. Likewise, unconnected outputs will remain on Reconfigurable Partition outputs, creating unnecessary waste in the overall design. These measures must be taken by the implementation tools to ensure that all Reconfigurable Modules have the same port map during design assembly.

It is strongly recommended that designers examine the interface of all Reconfigurable Partitions after synthesis to ensure that as few constants or unconnected ports as possible remain. By clearing out dead logic, resource utilization will be lower, and congestion and timing closure challenges can be made easier.

Six different cases are possible for partition interface usage:

1. **Both Static and Reconfigurable Module sides have active logic.** (Applies to partition inputs or outputs)

   This is the optimal situation. A partition pin will be inserted.

   However, if partition inputs are driven by VCC or GND, it is strongly recommended that the designer pushes these constants into the Reconfigurable Modules. This will reduce LUT usage and allow the implementation tools to optimize these constants with the RM logic.
2. **The Static side has an active driver but the Reconfigurable Module does not have active loads.** (Applies to partition inputs)

   This is acceptable, since this accommodates the scenario where not every Reconfigurable Module has the same I/O requirements. A partition pin will be inserted, and the unused input ports will be left unconnected.

   For example, one module may require CLK_A, while a second may require CLK_B. Clock spines will be pre-routed to the Reconfigurable Partition clock regions, but the module will only tap onto the clock source that is needed.

3. **The Static side has active loads but the Reconfigurable Module does not have an active driver.** (Applies to partition outputs)

   This is acceptable and similar to the case above. A partition pin will be inserted, and it will be driven by ground (logic 0) within the Reconfigurable Module.

4. **The Static side does not have an active driver, but the Reconfigurable Module has active loads.** (Applies to partition inputs)

   This will result in an error that must be resolved by modifying the partition interface.

   Here is an example of an error that may be seen for this scenario:

   ```plaintext
   ERROR: [Opt 31-65] LUT input is undriven either due to a missing connection from a design error, or a connection removed during opt_design.
   ```

   This error message would be followed by a LUT instance that is within the Reconfigurable Module.

5. **Reconfigurable Module has an active driver, but the Static side has no active loads.** (Applies to partition outputs)

   This will not result in an error, but is far from optimal as the RM logic will remain. No partition pin will be inserted.

6. **Neither Static nor Reconfigurable Module sides have driver or loads for a partition port.** (Applies to partition inputs or outputs)

   Nothing is inserted or used, so there is no implementation inefficiency, but it is unnecessary in terms of the instantiation port list.

---

**Global Clocking Rules**

Because the clocking information for every Reconfigurable Module for a particular Reconfigurable Partition is not known at the time of the first implementation, the PR tools pre-route each BUFG output driving a partition pin on that RP to all clock regions that the
Pblock encompasses. This means that clock spines in those clock regions might not be available for static logic to use, regardless of whether the RP has loads in that region.

In 7 series devices, up to 12 clock spines can be pre-routed into each clock region. This limit must account for both static and reconfigurable logic. For example, if 3 global clocks route to a clock region for static needs, any RP that covers that clock region can use the 9 global clocks available, collectively, in addition to those three top-level clocks.

In the example shown in Figure 3-1, icap_clk is routed to clock regions X0Y1, X0Y2, and X0Y3 prior to placement, and static logic is able to use the other clock spines in that region.

Figure 3-1: Pre-routing Global Clock to Reconfigurable Partition

If there are a large number of global clocks driving an RP, Xilinx recommends that area groups that encompass complete clock regions be created to ease placement and routing of static logic.
Partition Pin Placement

Each pin of an RP will have a partition pin (PartPin). By default the tools will automatically place these PartPins inside of the RP Pblock range (which is required). For many cases, this automatic placement may be sufficient for the design. However, for timing-critical interface signals or designs with high congestion, it may be necessary to help guide the placement of the PartPins. There are a few ways this can be done.

- Define user HD.PARTPIN_RANGE constraints for some or all of the pins.

```bash
set_property HD.PARTPIN_RANGE {SLICE_Xx0Yx0:SLICE_Xx1Yy1
SLICE_XxNYyN:SLICE_XxMYyM} [get_pins <rp_cell_name>/*]
```

By default the HD.PARTPIN_RANGE will be set to the entire Pblock range. Defining a user range allows the tools to place PartPins in the specified areas, improving timing and/or reducing congestion.

**IMPORTANT:** The automatic placement of PartPins will likely place PartPins in the corner of the Pblock rectangle. These corner locations have limited resources due to the CONTAIN_ROUTING property on the RP Pblock. If routing congestion exists around the corner of the Pblock, adding a specific HD.PARTPIN_RANGE that excludes the corner sites will help.

- By default, the tools place a maximum of 5 PartPins per interconnect (INT) tile. This can be adjusted via a param called `hd.maxPPLOCSPerIntTile`.

```bash
set_param hd.maxPPLOCSPerIntTile 3
```

Setting the value to 3 will tell the placer to put only 3 PartPins per INT tile, which will spread out the PartPin placement and reduce congestion around the RP interface.

**IMPORTANT:** If there are too many PartPins on the RP interface, the tools may have to violate the maximum number defined by this param. Always try to reduce the RP interface to a minimum set of pins.

Active Low Resets and Clock Enables

In the Xilinx 7 series architectures there are no local inverters on control signals (resets or clock enables). The following description uses a reset as the example, but the same applies for clock enables.

If a design uses an active low reset, a LUT must be used to invert the signal. In non-PR designs that use all active low resets multiple LUTs will be inferred, but can be combined into a single LUT and pushed into the I/O elements (the LUT goes away). In non-PR designs that use a mix of high and low, the LUT inverters can be combined into one LUT that remains
in the design, but that has minimal effect on routing and the timing of the reset net (output of LUT can still be put on global resources). However, for a design that uses active low resets on a partition, it is possible to have inverters inferred inside of the partition that cannot be pulled out and combined. This makes it impossible to put the reset on global resources, and can lead to poor reset timing and to routing issues if the design is already congested.

The best way to avoid this is to avoid using active low control signals. However, there are cases where this is not possible (for example, when using an IP core with an Advanced eXtensible Interface (AXI) interface). In these cases the design should assign the active-low reset to a signal at the top level, and use that new signal everywhere in the design.

As an example:

\[ \text{reset}_n <= \neg \text{reset}; \]

Use the \text{reset}_n signal for all cases, and do not use the \neg \text{reset} assignments on signals or ports.

This will ensure that a LUT will be inferred only for the reset net for the whole design, and will have a minimal effect on design performance.

---

### Decoupling Functionality

Because the reconfigurable logic is modified while the FPGA device is operating, the static logic connected to outputs of Reconfigurable Modules must ignore data from Reconfigurable Modules during Partial Reconfiguration. The Reconfigurable Modules will not output valid data until Partial Reconfiguration is complete and the reconfigured logic is reset. There is no way to predict or simulate the functionality of the reconfiguring module.

It is in the designer's hands to decide how the decoupling strategy is solved. A common design practice to mitigate this issue is to register all output signals (on the static side of the interface) from the Reconfigurable Module. An enable signal can be used to isolate the logic until it is completely reconfigured. Other approaches range from a simple 2 to 1 MUX on each output port, to higher level bus controller functions.

The static design should include the logic required for the data and interface management. It can implement mechanisms such as handshaking or disabling interfaces (which might be required for bus structures to avoid invalid transactions). It is also useful to consider the down-time performance effect of a PR module (that is, the unavailability of any shared resources included in a PR module during or after reconfiguration).
Reconfigurable Partition Pblock Sizes and Shapes

As noted in Apply Reset After Reconfiguration in Chapter 2, the height of the Reconfigurable Partition must align to clock region boundaries if RESET_AFTER_RECONFIG is to be used. Otherwise, any height may be selected for the Reconfigurable Partition.

The width of the Reconfigurable Partition must be set appropriately to make most efficient usage of interconnect and clocking resources. The left and right edges of Pblock rectangles should be placed between two resource columns (e.g. CLB-CLB, CLB-BRAM or CLB-DSP) and not between two interconnect columns (INT-INT). This allows the placer and router tools the full use of all resources for both static and reconfigurable logic. Implementation tool DRCs provide guidance if this approach is not followed.

In Figure 3-2, note that the left and right edges are drawn between CLB columns for the Pblock highlighted in white. Visualization of the interconnect tiles as shown in this image requires that the routing resources are turned on, using this symbol in the Device View:

![Image of interconnect tiles with routing resources turned on]

**Figure 3-2:** Optimal - Reconfigurable Partition Pblock Splitting CLB-CLB on Both Left and Right Edges

The Reconfigurable Partition Pblock must include all reconfigurable element types within the shape drawn. In other words, if the rectangle selected encompasses CLB (Slice), BRAM and DSP elements, all three types must be included in the Pblock constraints. If one of these...
is omitted, a DRC will be triggered alerting the fact that a split interconnect situation has been detected.

Other considerations must be taken if the Reconfigurable Partition spans non-reconfigurable sites, such as the center-column clocking resources or configuration components (ICAP, BSCAN, etc.), or abuts non-reconfigurable components such as I/O. If a Pblock edge splits interconnect columns for different resource types, implementation tools will accept this layout, but will restrict placement in the columns on each side of the boundary. If this poses issues with needed sites, the Pblock must be broken into multiple rectangles to clearly define reconfigurable logic usage.

For example, the initial floorplan shown in Figure 3-3 spans the center column, which contains clock buffer resources (BUFHCE). You can see these resources have not been included in the Pblock, as they are not highlighted in Figure 3-3.
Reconfigurable Partition Pblock Sizes and Shapes

The implementation tools will automatically prevent placement on both sides of the back-to-back interconnect sites along the center column, which will become an issue if those clock resources are needed for the design. To avoid this restriction, create multiple Pblock rectangles that avoid splitting interconnect columns, as shown in Figure 3-4.

Figure 3-5 is a close-up of this split, showing Slice (CLB) and Interconnect (INT) resource types. The gap between the two Pblock rectangles gives full access to the BUFHCE components to route completely using static resources. This also leaves one column of CLBs available for the static design to use.
Irregular shaped Partitions (such as a T or L shapes) are permitted, but users are encouraged to keep overall shapes a simple as possible. Placement and routing in such regions can become challenging, because routing resources must be entirely contained within these regions. Boundaries of Partitions can touch, but this is not recommended, as some separation helps mitigate potential routing restriction issues. Nested or overlapping Reconfigurable Partitions (Partitions within Partitions) are not permitted.

Finally, only one Reconfigurable Partition can exist per physical Reconfigurable Frame. A Reconfigurable Frame is the smallest size physical region that can be reconfigured, and aligns with clock region boundaries. A Reconfigurable Frame cannot contain logic from more than one Reconfigurable Partition. If it were to contain logic from more than one Reconfigurable Partition, it would be very easy to reconfigure the region with information from an incorrect Reconfigurable Module, thus creating contention. The Vivado® tools are designed to avoid that potentially dangerous occurrence.

### Black Boxes

The Partial Reconfiguration software does not yet allow black boxes to be implemented as Reconfigurable Modules directly. However, full and partial bit files can be generated after running `update_design -black_box` on Reconfigurable Partitions.

Even though a black box has no user logic contained in the logical representation of the design, the physical region is not entirely empty. Static routes that pass through the region,
including interface nets up to the partition pin nodes, exist within this region. Programming information for these signals is included in the black box programming bitstream.

Use of black boxes is an effective way to reduce the size of full configuration BIT file, and therefore reduce the initial configuration time. The compression feature may be enabled to reduce the size of BIT files. This option looks for repeated configuration frame structures to reduce the amount of configuration data that must be stored in the BIT file. This savings is seen in reduced configuration and reconfiguration time. When the compression option is applied to a routed PR design, all of the BIT files (full and partial) are created as compressed BIT files. To enable compression, set this property prior to running `write_bitstream`:

```
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
```

## Implementation Strategies

There are trade-offs associated with optimizing any FPGA design. Partial Reconfiguration is no different. Partitions are barriers to optimization, and reconfigurable frames require specific layout constraints. These are the additional costs to building a reconfigurable design. The additional overhead for timing and area needs vary from design to design. To minimize the impact, follow the design considerations stated in this guide.

When building Configurations of a reconfigurable design, the first Configuration to be chosen for implementation should be the most challenging one. Be sure that the physical region selected has adequate resources (especially elements such as BRAM and DSP48) for each Reconfigurable Module in each Reconfigurable Partition, then select the most demanding (in terms of either timing or area) RM for each RP. If all of the RMs in the subsequent Configurations are smaller or slower, it will be easier to meet their demands. Timing budgets should be established to meet the needs of all Reconfigurable Modules.

If it is not clear which reconfigurable module is the most challenging, each can be implemented in parallel in context with static, allowing static to be placed and routed for each. Examine resource utilization statistics and timing reports to see which configuration met design criteria most easily and which had the tightest tolerances, or which missed by the widest margins.

**IMPORTANT:** Focus attention on the configuration that is the furthest from meeting its goals, iterating on design sources, constraints, and strategies until needs are met. At some point, one configuration must be established as the golden result for the static design, and that implementation of the static logic will be used for all other configurations.

## Building Up Implementation Requirements

Implementation of Partial Reconfiguration designs requires that certain fundamental rules are followed. These rules have been established to ensure that a partial bitstream can be
Implementation Strategies

accurately created and safely delivered to an active FPGA. As noted throughout this
document, these rules include these basic premises:

- The logical and physical interface of a Reconfigurable Partition remains consistent as
each Reconfigurable Module is implemented.
- The logic and routing of a Reconfigurable Module is fully contained within a physical
region which will then be translated into a partial bitstream.
- The logic of the static design must be kept out of the reconfigurable region if the
dedicated initialization feature is used.

These requirements necessitate specific implementation rules for optimization, placement
and routing. Application of these rules may make it more difficult to meet design goals,
including timing closure. A recommended strategy is to build up this set of requirements
one at a time, allowing you to analyze the results at each step. Starting with the most
challenging configuration and the full set of timing constraints, implement the design
through place and route and examine the results, making sure you have sufficient timing
slack and resources available to continue to the next step.

1. First, implement the design with no Pblocks. Use bottom-up synthesis and follow
general Hierarchical Design recommendations, such as registered boundaries, to
achieve a baseline result.

2. Add Pblocks for the design partitions that will later be marked reconfigurable. This
floorplan can be based on the results established in the bottom-up synthesis run from
Step 1. Logic from the Reconfigurable Modules must be placed in the Pblocks, but static
logic may be included there as well.

   While creating these Pblocks, the HD.RECONFIGURABLE property (and optionally, the
   RESET_AFTER_RECONFIG property) can be added temporarily in order to run PR-specific
   Design Rule Checks. This will ensure that the floorplan created will meet PR size and
   alignment requirements.

3. With the floorplan established, separate the placement of static design resources from
those that will be reconfigurable by adding the EXCLUSIVE_PLACEMENT property to the
Pblocks. This will keep static logic placed outside the defined Pblocks.

4. Keep the routing for Reconfigurable Modules bound within the Pblocks by applying the
CONTAIN_ROUTING property to the Pblocks. With the properties from this and the
previous step, the only remaining rules relate to boundary optimization procedures as
well as PR-specific Design Rule Checks.

5. Finally, mark the Reconfigurable Partition Pblocks as HD.RECONFIGURABLE. The
EXCLUSIVE_PLACEMENT and CONTAIN_ROUTING properties are now redundant and can
be removed.

If design requirements are not met at any of these steps, you have to opportunity to review
the design structure and constraints in light of the newly applied implementation condition.
Design Revision Checks

A partial bitstream contains programming information and little else, as described in Chapter 4, Configuring the FPGA Device. While you do not need to identify the target location of the bitstream (the die location is determined by the addressing that is part of the BIT file), there are no checks in the hardware to ensure the partial bitstream is compatible with the currently operating design. Loading a partial bitstream into a static design that was not implemented with that Reconfigurable Module revision can lead to unpredictable behavior.

Xilinx suggests that you prefix a partial bitstream with a unique identifier indicating the particular design, revision and module that follows. This identifier can be interpreted by your configuration controller to validate that the partial bitstream is compatible with the resident design - a mismatch can be detected and the incompatible bitstream can be rejected before being loaded into configuration memory. This functionality must be part of your design, and would be similar to or in conjunction with decryption and/or CRC checks, as described in PRC/EPRC: Data Integrity and Security Controller for Partial Reconfiguration (XAPP887) [Ref 12].

A bitstream feature provides a simple mechanism for tagging a design revision. The BITSTREAM.CONFIG.USR_ACCESS property allows you to enter a revision ID directly into the bitstream. This ID is placed in the USR_ACCESS register, accessible from the FPGA fabric through a library primitive of the same name. Partial Reconfiguration designs can read this value and compare it to information in a partial bitstream to confirm the revisions of the design match. More information on this switch can be found in the Bitstream Settings Appendix in the Vivado Design Suite User Guide: Programming and Debugging (UG908) [Ref 13] and in Bitstream Identification with USR_ACCESS (XAPP497) [Ref 14].

CAUTION! Do not use the TIMESTAMP feature, since this value will not be consistent for each call to write_bitstream. Only select a consistent, explicit ID to be used for all write_bitstream runs.

Simulation and Verification

Configurations of Partial Reconfiguration designs are complete designs in and of themselves. All standard simulation, timing analysis, and verification techniques are supported for PR designs. Partial reconfiguration itself cannot be simulated.
Chapter 4

Configuring the FPGA Device

Configuration Overview

This chapter describes the system design considerations when configuring the FPGA device with a partial BIT file, as well as architectural features in the FPGA that facilitate Partial Reconfiguration. Because most aspects of Partial Reconfiguration are no different than standard full configuration, this section concentrates on the details that are unique to PR.

Any of the following configuration ports can be used to load the partial bitstream: SelectMAP, Serial, JTAG, or ICAP (Internal Configuration Access Port). For Zynq®-7000 AP SoC devices, deliver partial bitstreams via the JTAG or PCAP (Processor Configuration Access Port) ports.

Note: If you need to partially reconfigure a Zynq device via the ICAP, please contact Xilinx support.

To use SelectMAP or Serial modes for loading a partial BIT file, these pins must be reserved for use after the initial device configuration. This is achieved by using the BITSTREAM.CONFIG.PERSIST property to keep the dual-purpose I/O for configuration usage and to set the configuration width. The details are documented in the Vivado Design Suite User Guide: Programming and Debugging (UG908) [Ref 13] and the syntax is:

```
set_property BITSTREAM.CONFIG.PERSIST <value> [current_design]
```

where <value> is one of the following: No, Yes, CTLReg, X1, X8, X16, X32, SPI1, SPI2, SPI4, BPI8, BPI

Partial bitstreams contain all the configuration commands and data necessary for Partial Reconfiguration. The task of loading a partial bitstream into an FPGA does not require knowledge of the physical location of the RM because configuration frame addressing information is included in the partial bitstream. A valid partial bitstream cannot be sent to the wrong part of the FPGA device.

A Partial Reconfiguration controller retrieves the partial bitstream from nonvolatile memory, then delivers it to a configuration port. The Partial Reconfiguration control logic can either reside in an external device (for example, a processor) or in the fabric of the FPGA device to be reconfigured. A user-designed internal PR controller loads partial bitstreams through the ICAP interface. As with any other logic in the static design, the internal Partial
Reconfiguration control circuitry operates without interruption throughout the Partial Reconfiguration process.

Internal configuration can consist of either a custom state machine, or an embedded processor such as MicroBlaze™. For a Zynq-7000 AP SoC, the Processor Subsystem (PS) can be used to manage Partial Reconfiguration events. Note that for Zynq-7000 devices, the Programmable Logic (PL) can be partially reconfigured, but the Processing System cannot.

As an aid in debugging Partial Reconfiguration designs and PR control logic, the Vivado® Hardware Manager tool can be used to load full and partial bitstreams into an FPGA device by means of the JTAG port.

For more information on loading a bitstream into the configuration ports, see the Configuration Interfaces chapter in these documents:

- 7 Series FPGAs Configuration User Guide (UG470) [Ref 1]
- Zynq-7000 AP SoC Technical Reference Manual (UG585) [Ref 13]

**Configuration Modes**

Partial Reconfiguration is supported using the following configuration modes:

- **ICAP** - A good choice for user configuration solutions. Requires the creation of an ICAP controller as well as logic to drive the ICAP interface.
- **PCAP** - An alternative configuration mechanism for Zynq-7000 designs.
- **JTAG** - A good interface for quick testing or debug. Can be driven using the Vivado Hardware Manager or ChipScope™ Analyzer using a Xilinx configuration cable that supports JTAG.
- **Slave SelectMAP** or **Slave Serial** - A good choice to perform full configuration and Partial Reconfiguration over the same interface.

Master modes are not directly supported due to IPROG housecleaning that will clear the configuration memory.
Downloading a Full Bit File

The FPGA device in a digital system is configured after power on reset by downloading a full BIT file either directly from a PROM or from a general purpose memory space by a microprocessor. A full BIT file contains all the information necessary to reset the FPGA device, configure it with a complete design and verify that the BIT file is not corrupt. Figure 4-2 illustrates this process.

Figure 4-2: Configuring With a Full Bit File

After the initial configuration is completed and verified, the FPGA device enters user mode, and the downloaded design begins functioning. If a corrupt BIT file is detected, the DONE signal is never asserted, the FPGA device never enters user mode, and the corrupt design never starts functioning.
# Downloading a Partial Bit File

A partially reconfigured FPGA device is in user mode while the partial BIT file is loaded. This allows the portion of the FPGA logic not being reconfigured to continue functioning while the reconfigurable portion is modified. Figure 4-4 illustrates this process.

![Configuration Mode](image)

**Figure 4-4:** Configuring With a Partial Bit File

The partial BIT file has a simplified header, and there is no startup sequence that brings the FPGA device into user mode. The BIT file contains (essentially, and with default settings) only frame address and configuration data, plus a final checksum value. Additional CRC checks can be inserted if desired, to perform bitstream integrity checking.

If Reset After Reconfiguration is selected, the DONE pin will pull low when reconfiguration begins, and pull high again when partial reconfiguration successfully completes, although the partial bitstream can still be monitored internally as well.

If Reset After Reconfiguration is not selected, you must monitor the data being sent to know when configuration has completed. The end of a partial BIT file has a DESYNCH word (0000000D) that informs the configuration engine that the BIT file has been completely delivered. This word is given after a series of padding NO OP commands, ensuring that once the DESYNCH has been reached, all the configuration data has already been sent to the target frames throughout the device. As soon as the complete partial BIT file has been sent to the configuration port, it is safe to release the reconfiguration region for active use.
System Design for Configuring an FPGA Device

A partial BIT file can be downloaded to the FPGA device in the same manner as a full BIT file. An external microprocessor determines which partial BIT file should be downloaded, where it exists in an external memory space, and directs the partial BIT file to a standard FPGA configuration port such as JTAG, Select MAP or serial interface. The FPGA device processes the partial BIT file correctly without any special instruction that it is receiving a partial BIT file.

It is common to assert the INIT or PROG signals on the FPGA configuration interface before downloading a full BIT file. This must not be done before downloading a partial BIT file, as that would indicate the delivery of a full BIT file, not a partial one.

Any indication to the working design that a partial BIT file will be sent (such as holding enable signals and disabling clocks) must be done in the design, and not by means of dedicated FPGA configuration pins. Figure 4-6 shows the process of configuring through a microprocessor.

In addition to the standard configuration interfaces, Partial Reconfiguration supports configuration by means of the Internal Configuration Access Port (ICAP). The ICAP protocol is identical to SelectMAP and is described in the Configuration User Guide for the target device. The ICAP library primitive can be instantiated in the HDL description of the FPGA design, thus enabling analysis and control of the partial BIT file before it is sent to the device.
configuration port. The partial BIT file can be downloaded to the FPGA device through general purpose I/O or gigabit transceivers and then routed to the ICAP in the FPGA fabric.

The ICAP must be used, with an 8-bit bus only, for Partial Reconfiguration for encrypted 7 series BIT files. Reconfiguration through external configuration ports is not permitted when encryption is used.

---

**Partial Bitstream CRC Checking**

A partial bitstream is loaded into an active design, and the default built-in CRC check does not occur until the end of the bitstream. However, the configuration engine has the ability to perform a frame-by-frame CRC check and will not load a frame into the configuration memory if that CRC check fails. A failure is reported on the INIT_B pin (it is pulled low) and gives you the opportunity to take the next step: retry the partial bit file, fall back to a golden partial bit file, etc. The partially loaded reconfiguration region will not have valid programming in it, but the CRC check ensures the remainder of the device stays operational while the system recovers from the error.

This per-frame CRC check will be supported in the next release of the Vivado® Design Suite.

After a partial bit file has been loaded (with or without the per-frame CRC checks), the overall configuration of the device has changed. If the POST_CRC feature for SEU mitigation is enabled, the SEU mitigation engine will automatically recalculate the embedded SEU CRC value after partial bitstream has been loaded and the configuration interface has been desynched by the user. Upon completion of the CRC recalibration, FRAME_ECCE2’s FRAME_VALID output will again toggle to indicate SEU detection has resumed.

---

**Configuration Time**

The speed of configuration is directly related to the size of the partial BIT file and the bandwidth of the configuration port. The different configuration ports in 7 series architectures have the maximum bandwidths shown in Table 4-1.

<table>
<thead>
<tr>
<th>Configuration Mode</th>
<th>Max Clock Rate</th>
<th>Data Width</th>
<th>Maximum Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICAP</td>
<td>100 MHz</td>
<td>32 bit</td>
<td>3.2 Gbps</td>
</tr>
<tr>
<td>SelectMAP</td>
<td>100 MHz</td>
<td>32 bit</td>
<td>3.2 Gbps</td>
</tr>
<tr>
<td>Serial Mode</td>
<td>100 MHz</td>
<td>1 bit</td>
<td>100 Mbps</td>
</tr>
<tr>
<td>JTAG</td>
<td>66 MHz</td>
<td>1 bit</td>
<td>66 Mbps</td>
</tr>
</tbody>
</table>
The exact bitstream length is available in the created .rbt file by using the -raw_bitfile option for write_bitstream. Use this number along with the bandwidth to calculate the total configuration time. Here is an example of the header in a raw bit file:

Xilinx ASCII Bitstream
Created by Bitstream 2013.4
Design name: led_shift_count;UserID=0xFFFFFFFF
Architecture:kintex7
Part: 7k325tffg900
Date: Thu Aug 29 16:42:05 2013
Bits: 1211072
11111111111111111111111111111111

Configuration Debugging

The ICAP interface can be used to monitor the configuration process, even if other configuration means are used (JTAG or Slave SelectMAP). In fact, the status of the configuration is automatically pushed out to the “O” port of the ICAP without having to issue a read.

The “O” port of the ICAP block is a 32-bit bus, but only the lowest byte is used. The mapping of the lower byte is as follows:

Table 4-2: ICAP “O” Port Bits

<table>
<thead>
<tr>
<th>ICAP “O” Port Bits</th>
<th>Status Bit</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| O[7]               | CFGERR_B   | Configuration error (active Low)  
0 = A configuration error has occurred.  
1 = No configuration error. |
| O[6]               | DALIGN     | Sync word received (active High)  
0 = No sync word received.  
1 = Sync word received by interface logic. |
| O[5]               | RIP        | Readback in progress (active High)  
0 = No readback in progress.  
1 = A readback is in progress. |
| O[4]               | IN_ABORT_B | ABORT in progress (active Low)  
0 = Abort is in progress.  
1 = No abort in progress. |
| O[3:0]             | 1          | Reserved |

The most significant nibble of this byte reports the status. These Status bits indicate whether the Sync word been received and whether a configuration error has occurred. The following table displays the values for these conditions.
Figure 4-8 shows a completed full configuration, followed by a partial reconfiguration with a CRC error, and finally a successful partial reconfiguration. Using the table above, and the description below, you can see how the “O” port of the ICAP can be used to monitor the configuration process. If a CRC error occurs, these signals can be used by a configuration state machine to recover from the error. These signals can also be used by ChipScope to capture a configuration failure for debug purposes. With this information ChipScope can also be used to capture the various points of a partial reconfiguration.

Table 4-3: ICAP Sync Bits

<table>
<thead>
<tr>
<th>O[7:0]</th>
<th>Sync Word?</th>
<th>CFGERR?</th>
</tr>
</thead>
<tbody>
<tr>
<td>9F</td>
<td>No Sync</td>
<td>No CFGERR</td>
</tr>
<tr>
<td>DF</td>
<td>Sync</td>
<td>No CFGERR</td>
</tr>
<tr>
<td>5F</td>
<td>Sync</td>
<td>CFGERR</td>
</tr>
<tr>
<td>1F</td>
<td>No Sync</td>
<td>CFGERR</td>
</tr>
</tbody>
</table>

The markers in the ChipScope display indicate the following:

- **1st_done**

  This marker indicates the completion of the initial full bitstream configuration. The DONE pin (done_pad in this waveform) goes High.
• **cfgerr**
  This marker indicates a CRC error is detected while loading partial bitstream. The status can be observed through O[31:0] (icap_o_top[31:0] in the waveform).
  - *Icap_o_top[31:0]* starts at 0x9F
  - After seen SYNC word, *Icap_o_top[31:0]* change to 0xDF
  - After detect CRC error, *Icap_o_top[31:0]* change to 0x5F for one cycle, and then switches to 0x1F
  - INIT_B pin is pulled low (*init_pad* in the waveform)

• **RCRC**
  This marker indicates when the partial bitstream is loaded again. The RCRC command resets the **cfgerr** status, and removes the pull-down on the INIT_B pin (*init_pad* in this waveform).
  - *Icap_o_top[31:0]* change from 0x1F to 0x5F when the SYNC word is seen
  - *Icap_o_top[31:0]* change from 0x5F to 0xDF when RCRC command is received

• **pr_done**
  This marker indicates a successful Partial Reconfiguration.
  - *Icap_o_top[31:0]* change from 0xDF to 0x9F when the DESYNC command is received and no configuration error is detected.
Chapter 5

Known Issues and Limitations

Known Issues

This is a list of issues that may be encountered when using Partial Reconfiguration in the Vivado® 2013.4 release. If you encounter any of these issues, or discover any others, please inform Xilinx and send an example design that shows the issue. These test cases are very helpful for our efforts to improve the overall solution.

- Please report to Xilinx® all cases of fatal or internal errors, incomplete routing (partial antennas), or other rule violations that prevent place and route, pr_verify, and write_bitstream from succeeding. Including a design showing the failure is critical for proper analysis and implementation of fixes.

- Reuse of implemented Reconfigurable Modules is not 100% preserved. In a future release, a checkpoint representing an implemented Reconfigurable Module could be saved from one configuration and then reused in another configuration. However, in the current release, the interface nets between the partition pins and the internal logic are not captured, so these signals must be rerouted.
  - This can be done by running route_design after loading in a routed RM checkpoint. This process has not been extensively tested and is not recommended.

- If the initial configuration of an SSI device (7V2000T, 7VX1140T) is done through an SPI interface, partial bitstreams cannot be delivered to the master (or any) ICAP; they must be delivered to an external port, such as JTAG. If the initial configuration is done through any other configuration port, the master ICAP can be used as the delivery port for partial bitstreams.
  - A workaround is being investigated and will be delivered in a future release.

- Do not drive multiple outputs of a single Reconfigurable Module with the same source. Each output of an RM must have a unique driver.

- After performing interactive floorplanning in the Vivado IDE, you may see unintended prohibited sites near the Reconfigurable Partition after place_design. These sites appear in the IDE as red circles with slashes. To avoid these prohibited sites, once the design Pblocks have been created and saved as described in Floorplan in the Vivado IDE in Chapter 2, simply close and re-open the design before continuing to opt_design or place_design.
Known Limitations

Certain features are not yet developed or supported in Vivado 2013.4. Some of these features may be added in upcoming releases. These include:

- When selecting Pblock ranges to define the size and shape of the Reconfigurable Partition, do not use the CLOCKREGION resource type. Pblock ranges must only include types SLICE, RAMB18, RAMB36, and DSP48 resource types.
- Black box support during implementation. Black boxes must be created post-implementation using `update_design -black_box`, since true black box support (implementing a configuration with an empty RM) is not yet supported.
  - Black boxes created via `update_design -black_box` are truly empty from a function perspective; no constant drivers are placed on outputs. Because of this, if bitstreams are used keep decoupling logic active to ensure any floating RM outputs do not disrupt the static design.
- Project support. Compiling configurations using projects and project commands (`create_run`, `launch_runs`, etc.) is not yet supported. Managing PR projects in the Vivado IDE is likewise not yet supported.
  - Checkpoints can be opened in the IDE and many analysis features can be used, but the Design Runs features cannot be used.
- The encryption and per-frame CRC check features for `write_bitstream` have not yet been implemented. These features are scheduled to be added in a future software release.
- Do not use Vivado Debug core insertion features within Reconfigurable Partitions. This flow inserts the debug hub, which includes BSCAN and BUFG primitives, which are not permitted inside reconfigurable bitstreams.
- Do not use Partial Reconfiguration with Tandem Configuration capabilities within Xilinx PCIe IP. These two solutions have not yet been tested together.
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:


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

1. 7 Series FPGAs Configuration User Guide (UG470)
3. Partial Reconfiguration User Guide (UG702) - For ISE Design Tools
4. Hierarchical Design Methodology Guide (UG748) - For ISE Design Tools
5. Repeatable Results with Design Preservation (WP362) - For ISE Design Tools
8. XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices (UG687)
10. 7 Series FPGAs GTX/GTH Transceivers User Guide (UG476)
11. MMCM and PLL Dynamic Reconfiguration (7 Series) (XAPP888)
12. PRC/EPRC: Data Integrity and Security Controller for Partial Reconfiguration (XAPP887)
14. Bitstream Identification with USR_ACCESS (XAPP497)