

Verification Continuum™

HAPS® Prototyping

User Guide

April 2022

SYNOPSYS®

solvnetplus.synopsys.com
Synopsys Confidential Information

Copyright Notice and Proprietary Information

© 2022 Synopsys, Inc. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

Free and Open-Source Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at

<http://www.synopsys.com/Company/Pages/Trademarks.aspx>.

All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
690 East Middlefield Road
Mountain View, CA 94043
www.synopsys.com

April 2022

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

Contents

Chapter 1: Starting with FPGA-Based Prototyping

| | |
|---|----|
| Introduction to FPGA-Based Prototyping | 16 |
| About the HAPS Prototyping Solution | 16 |
| HAPS ProtoCompiler and ProtoCompiler DX Functionality | 17 |
| HAPS ProtoCompiler Design Flows | 19 |
| HAPS ProtoCompiler DX Design Flows | 22 |
| HAPS Prototyping and Standard Prototyping Use Models | 24 |
| Starting the Tool | 27 |

Chapter 2: Converting ASIC Designs to FPGA

| | |
|---|----|
| Getting ASIC Designs Ready for FPGA | 34 |
| Dividing up the Design | 36 |
| Handling I/O Pads | 37 |
| Handling User-Defined Primitives | 37 |
| Converting ASIC Memories | 38 |
| Making ASIC Netlists Suitable for FPGA Design | 41 |
| Converting ASIC Clocks | 42 |
| General Guidelines for ASIC Clocks | 42 |
| Defining Clocks | 44 |
| Scaling External Clock Speed | 47 |
| Dealing with IP | 53 |
| Working with Power Specifications | 54 |
| Converting Constraints | 55 |
| Checking Resources | 55 |

Chapter 3: Creating and Compiling Databases

| | |
|---|-----|
| Working with a Design Database | 58 |
| Creating a Design Database | 58 |
| Loading and Navigating Database States | 62 |
| Converting Legacy Projects to Databases | 65 |
| Archiving and Unarchiving Databases | 66 |
| Compiling the Design | 77 |
| Compiling with the run compile Command | 80 |
| Running Diagnostic Compiler Mode | 85 |
| Running Fast Compiler Mode (Standard Compiler) | 86 |
| Using Different Standard Compiler Modes Together | 87 |
| Running Bottom-Up Compile | 90 |
| Controlling the Compiler Run with Options and Constraints | 92 |
| Adding Design Files | 96 |
| Specifying Source Files (Standard Compiler) | 97 |
| Specifying Source Files Using a Command Argument | 97 |
| Creating a Text File List of Source Files | 101 |
| Specifying Source Files from the GUI | 103 |
| Checking for Unused Files | 105 |
| Using Variables in File Paths | 105 |
| Specifying Verilog Standards | 107 |
| Using UUM and Group Mapping | 109 |
| Creating an Imf File | 110 |
| Setting Library Options with -hdl_define | 114 |
| Using Unified Compile | 120 |
| Unified Compile Design Flow | 120 |
| Running Unified Compile | 121 |
| Preparing the Input for Unified Compile | 125 |
| Setting up the RTL Design Files and Library Mapping | 127 |
| Setting up the UTF File and the VCS Script | 128 |
| Including IPs in a UC Design | 130 |
| Including Complex SystemVerilog Port Modules | 133 |
| Checking the UC Database | 135 |
| Running zFmCheck to Validate VCS Design Files | 135 |
| Running Lint Checks with VC Static | 137 |
| Checking for CDC Violations with VC Static | 139 |
| Compiling Incrementally | 144 |
| Design Database to SRS Incremental Flow (UC) | 144 |

| | |
|--|-----|
| Compiling Incrementally Using Bypass Flow (UC) | 146 |
| Compiling Incrementally Using Structural VM (UC) | 156 |
| Running Bottom-Up Compile (UC) | 159 |
| Using the Bypass Flow (Standard Compiler) | 165 |
| Specifying Constraints | 167 |
| Adding Constraints to a Design Database | 167 |
| Specifying Directives in a CDC File | 168 |
| Setting Options | 171 |
| Chapter 4: Partitioning the Design | |
| Overview of Partitioning | 178 |
| The Partitioning Flow | 178 |
| Design Methodology for Partitioning | 179 |
| Setting up Files for Partitioning | 183 |
| Defining the System in a TSS File | 184 |
| Exploring Partition Choices with a Basic TSS | 185 |
| Defining the Target System in a Detailed TSS File | 189 |
| Defining Clocks in the TSS File | 194 |
| Defining Hardware Interconnect | 196 |
| Defining Connections with Auto-Cabling | 197 |
| Defining Connections with <code>board_system_create -interconnect</code> | 200 |
| Defining Daughter Boards | 204 |
| Moving from Abstract PCF to TSS: Flow Example | 220 |
| Validating TSS Cable Connections with <code>conspeed_hstdm</code> | 225 |
| Validating the TSS Against the Hardware | 226 |
| Generating TSS Files Using TSS Builder | 228 |
| Choosing Systems and Daughter Boards | 229 |
| Chaining CDEs | 232 |
| Assigning Clocks | 233 |
| Assigning Top IO | 235 |
| Connecting Cables Within and Across Systems | 236 |
| Connecting Daughter Boards | 237 |
| Generating TSS Files | 238 |
| Defining Constraints in PCF Files | 241 |
| Using Abstract PCF Commands for Partition Exploration | 242 |
| Specifying PCF Constraints for Partitioning | 245 |
| Specifying PCF Constraints in the PCF Editor (GUI) | 254 |
| Using PCF Queries for TSS and Netlist Information | 261 |
| Manually Assigning Inter-FPGA Nets to Traces | 265 |

| | |
|--|-----|
| Working Iteratively with PCF Assignments | 267 |
| Working with HAPS Systems | 269 |
| Working with HAPS Global Clocks | 269 |
| Assigning and Distributing HAPS-80 Reset | 278 |
| Defining HAPS-80 Hardware | 283 |
| Defining a HAPS-DX7 System as a Subsystem | 289 |
| Defining HAPS-80 Desktop Single/Dual (HAPS-80D) | 291 |
| Working with HAPS-100 Modules | 296 |
| Working with HAPS-100 Clocks | 296 |
| Generating Controlled Clocks using HAPS Clock Generator | 308 |
| Working with HAPS-100 Resets | 318 |
| Using UMRBus 3.0 with HAPS-100 Modules | 320 |
| Using run pre_partition to Define Partitions | 324 |
| Partitioning the Logic | 328 |
| Using run partition to Create Partitions | 329 |
| Partitioning Single-FPGA Designs | 335 |
| Working with Area Estimates for Partitioning | 337 |
| Using Timing-Aware Partitioning | 341 |
| Optimizing Multi-Hop Paths | 344 |
| Partitioning Clocks | 348 |
| Locking Down Partitions | 352 |
| Using Interactive Partitioning | 355 |
| Analyzing Partition Result Files | 358 |
| Checking the Target Specification (TSS) | 358 |
| Methodology for Analyzing Partitioning at Different Stages | 360 |
| Checking Partitioning Reports | 367 |
| Parsing Partitioning Reports | 370 |
| Using Time Domain Multiplexing (TDM) | 372 |
| Setting General Controls for TDM | 373 |
| Using HSTDMD and HSTDMD ERD | 376 |
| Working with HSTDMD Training Methods | 377 |
| Using proto_rt for HSTDMD Training | 377 |
| Using HAPS-Controlled HSTDMD Training | 380 |
| Using the Reset Hijack Method for HSTDMD Training | 382 |
| Estimating System Frequency Based on HSTDMD Delays | 384 |
| Analyzing HSTDMD Results | 395 |
| Using HSTDMD Top I/O Flow | 397 |
| Using Asynchronous Pin Multiplexing (ACPM) | 400 |
| Using Multi-Gigabyte Transceivers for TDM | 401 |

| | |
|---|-----|
| Using Hierarchical Partitioning | 406 |
| Running Hierarchical Partitioning in Automatic Single-Pass Mode | 408 |
| Running Top-Down Hierarchical Partitioning in Manual Mode | 409 |
| Running System Route for the Top Level | 412 |
| Congestion Reduction Using Pin Table Assignment | 414 |
| Generating FPGAs | 416 |
| Using system_generate to Generate FPGAs | 416 |
| Analyzing Results after System Generate | 418 |
| Interactive Partitioning Flow | 421 |
| Working with Inferred Clocks Driving Inter-FPGA Paths | 424 |
| Implementing Individual FPGAs | 426 |
| Running Synthesis for Individual FPGAs | 426 |
| Customizing Scripts to Synthesize, Place, and Route FPGAs | 430 |
| Updating the Top Level with FPGA Implementation Results | 435 |
| Using Multi-Design Mode (MDM) | 437 |
| Running Designs in Multi-Design Mode: HAPS-100 | 437 |
| Running MDM in a Mixed-Design Setup: HAPS-100 and HAPS-80 | 442 |
| Running MDM in a Mixed-Design Setup: HAPS-80 and HAPS-70 | 444 |
| MDM Planning Guidelines for HAPS-70 and HAPS-80 | 446 |
| Instantiating CAPIM_UI | 452 |
| Using UMRBus 3.0 with HAPS-100 Modules | 455 |
| Running Designs in Multi-Design Mode: HAPS-80 and HAPS-70 | 458 |
| MDM Planning Guidelines (HAPS-80 and HAPS-70) | 461 |
| Instantiating CAPIM_UI | 467 |

Chapter 5: Running the Implementation Flow

| | |
|---|-----|
| The Basic Implementation Flow | 472 |
| Running Pre-Map | 474 |
| Mapping to Hardware | 481 |
| Mapping the Design | 490 |
| Synthesizing Based on Design Intent | 493 |
| Using Distributed Processing | 497 |
| Setting Options to Run Distributed Processing | 498 |
| Using CDPL for Distributed Processing | 500 |
| Running Distributed Compile | 503 |
| Running Distributed Synthesis | 506 |
| Analyzing Results | 512 |

| | |
|---|-----|
| Checking Reports and Log Files | 512 |
| Viewing Results in a Schematic | 517 |
| Exploring a Schematic with find and expand | 520 |
| Querying Incremental Results | 520 |
| Querying Jobs | 523 |
| Querying Metrics for a Design | 524 |
| Dealing with Black Boxes in the Synthesized Netlist | 526 |
| Checking Timing Results | 528 |
| Generating and Viewing Timing Reports | 528 |
| Viewing and Correlating Results with the Timing Report View | 535 |
| Generating Custom Timing Reports with the Timing Analyst | 541 |
| Checking Clock Information | 546 |
| Running System-Level Timing Analysis (SLTA) | 547 |
| Handling Errors and Warnings | 550 |
| Working with Errors and Warnings | 550 |
| Manipulating Message Display and Reporting | 551 |
| Running Simulation | 557 |
| Setting up the Tools and Testbench | 558 |
| Running Gate-Level Simulation for a Single-FPGA Design | 560 |
| Running RTL Simulation for a Multi-FPGA Design | 562 |
| Simulating Post-Partition Designs | 567 |
| Running Gate-Level Simulation for a Multi-FPGA Design | 573 |
| Running DUT/IP Separation Simulation Flow for HAPS-100 | 575 |
| Generating a Testbench from FSDB | 575 |
| Running Place and Route | 576 |
| Running Place-and-Route Exploration | 576 |
| Running Place and Route without Exploration | 580 |
| Customizing Scripts to Run Vivado (Single-FPGA Designs) | 583 |
| Importing Place and Route Results for Backannotation | 584 |
| Resynthesizing After Place and Route | 586 |
| Analyzing Congestion Issues | 589 |
| Using Congestion-Reducing Techniques | 589 |
| Troubleshooting Congestion at Different Design Phases | 591 |
| Using TDM to Reduce SLL Congestion in Vivado | 598 |
| Working with Mixed-System Designs | 601 |
| Running in Batch Mode | 603 |
| Running Batch Mode with a Tcl Script | 603 |
| Queueing Licenses | 604 |
| Releasing the Tool License During Place and Route | 607 |

| | |
|--|-----|
| Working with Tcl Scripts and Commands | 609 |
| Using Tcl Commands and Scripts | 609 |
| Generating a Tcl Script from the GUI | 609 |
| Creating a Tcl Script | 611 |
| Setting Number of Parallel Jobs | 612 |
| Preparing to Run on the Hardware | 614 |
| Exporting Files for Runtime | 614 |
| Configuring Bit Files | 616 |
| Generating a Confpro Project | 617 |
| Ensuring Hardware Bring-up | 621 |
| Configuring HAPS Clocks | 624 |
| Running HAPS Hardware Diagnostics | 626 |
| Running Data Expansion in a Multi-User Setup | 627 |

Chapter 6: Instrumenting and Debugging Designs

| | |
|---|-----|
| Evaluating Debug Methodology Choices | 634 |
| The Instrumentor-Debugger Flow | 638 |
| Instrumenting the Design for Debug | 641 |
| Instrumenting the Design Before Compiling | 642 |
| Instrumenting the Design After Compiling | 648 |
| Defining IICE Parameters | 651 |
| Selecting Buffer Type | 657 |
| Specifying IICE Clocks | 659 |
| Making Incremental Instrumentation Changes | 663 |
| Making Incremental Changes in UC Instrumentation | 665 |
| Instrumenting Multiple Clocks in a Single IICE | 668 |
| Capturing Startup Errors with Pre-Armed Triggers | 670 |
| Instrumenting with Unified Compile | 672 |
| Verifying UC Designs | 677 |
| Debugging Designs with SVAs | 677 |
| Running Incremental Unified Debug | 678 |
| Running the RTL Debugger | 681 |
| Launching and Running the Debugger | 681 |
| Debugging a Multi-Clock Design | 684 |
| Varying Buffer Depth Dynamically for DDR3 | 687 |
| Viewing Captured Deep Trace Debug Samples | 689 |
| Verifying the Hardware Configuration for Deep Trace Debug | 690 |
| Configuring CEL-Based Triggers | 692 |
| Working with Deep Trace Debug | 695 |

| | |
|---|-----|
| Running Deep Trace Debug (DTD) | 702 |
| Running Single-FPGA Debug on a HAPS-80 FPGA | 703 |
| Running Single-FPGA Debug on HAPS-70 with DDR3 | 707 |
| Running Multi-FPGA Built-in DTD (HAPS-80) | 711 |
| Running Multi-FPGA Debug with HAPS-80 Built-in Memory | 712 |
| Using a Debug Hub with SATA (DTD2) | 716 |
| Multi-FPGA Debug Hub Flow for HAPS-70 Systems | 717 |
| Running DTD2 with a HAPS-DX7 System and SATA Cabling | 718 |
| Configuring the Sample Clock for DTD2 | 723 |
| Working with Maximum Sample Frequency for DTD2 | 724 |
| Configuring a Reference Clock for DTD2 | 725 |
| Configuring Clocks on the HAPS-DX7 System for DTD2 | 727 |
| Using Xilinx Backbone Clock Routes with DTD2 | 727 |
| Configuring Resets for DTD2 | 728 |
| Allocating Locations for PLLs and MMCMs | 730 |
| Setting up Hardware for DTD2 with HAPS-70 | 731 |
| Connecting the HAPS-70 and HAPS-DX7 Systems for DTD2 | 731 |
| Using SATA Cable MGT Links | 734 |
| Reserving MGT Links | 736 |
| Setting up Fixed Debug Cable Connections | 737 |
| Chaining HAPS-70 Systems for Debug | 738 |
| Using a Debug Hub with QSFP Connectors | 740 |
| Using BRAM for Debugging | 743 |
| Using Real-time Debugging | 744 |
| Running Incremental Debug for ECOs | 748 |
| Using Global State Visibility (GSV) | 753 |
| Asynchronous and Synchronous GSV | 754 |
| Using Asynchronous GSV for a Single-FPGA Design | 757 |
| Use Model Example: Synchronous GSV for a Single-FPGA Design | 760 |
| Use Model Example: Synchronous GSV with a Multi-FPGA Design | 762 |
| Instantiating and Configuring Clock Control Modules for GSV | 764 |
| Memory GSV | 768 |
| Example: Implementing Memory GSV for a Single-FPGA Design | 768 |
| Setting up the Hardware for GSV | 772 |
| Running GSV with Clock Control | 773 |
| GSV or eGSV Flow Using ZCEI based CCM on HAPS-100 | 776 |
| GSV/eGSV Flow Using HAPS-100 Clock Generator | 785 |
| Integrated Debugging with the Verdi Software | 792 |

| | |
|---|-----|
| Using Verdi with the Standard Compiler | 793 |
| Using Verdi and Siloti for Debugging | 797 |
| Using the Verdi Flow with a UC Design | 803 |
| Running Formal Verification with Checkpoints | 811 |
| Chapter 7: Migrating Designs for Synthesis | |
| Using Continue on Error | 820 |
| Running Continue on Error During Compilation | 821 |
| Analyzing Compilation Errors After Continue on Error | 822 |
| Running Continue on Error During Mapping | 826 |
| Including UPF Specifications | 827 |
| Incorporating UPF in the Standard Compiler Flow | 828 |
| Specifying UPF Power Domains | 830 |
| Specifying Isolation Cell Strategies | 835 |
| Modeling Retention Logic | 847 |
| Checking the UPF Implementation | 857 |
| Verifying Designs with UPF Information | 860 |
| Converting Gated Clocks | 862 |
| Overview of Clock Conversion | 862 |
| Working with Gated and Generated Clocks | 870 |
| Defining Clocks for Gated Clock Conversion | 872 |
| Interpreting Gated Clock Error Messages | 895 |
| Conversion of Datapaths with Latches | 905 |
| Editing Netlists | 907 |
| Editing Netlists with HAPS ProtoCompiler Commands | 907 |
| Editing a Verilog Netlist | 910 |
| Using Gate-Level ASIC Netlists | 911 |
| Creating Parallel Code with force and bind Statements | 911 |
| Using Cross-Module Referencing (XMR) | 913 |
| Cross-Module Referencing Using a CDC Constraint | 913 |
| Defining Cross-Module References in the HDL | 914 |
| Using Hyper Source Modules for XMR | 915 |
| Converting Designs for Unified Compile | 920 |
| Converting Older Designs for Unified Compiler | 920 |
| Handling HDL Differences | 921 |
| Chapter 8: Verifying the Design | |
| Guidelines for Successful Formal Verification | 938 |

| | |
|--|-----|
| Verifying Single-FPGA Designs | 945 |
| Verifying Multi-FPGA Designs | 953 |
| Setting Options to Guide Formal Verification | 958 |
| Running VCS Simulation for Inferred Memories | 960 |
| Running Formal Verification with Checkpoints | 962 |

Chapter 9: Recommended Methodologies

| | |
|--|-----|
| Methodology for a Quick Prototype | 966 |
| Setting up the FPGA Design and Compiling It | 967 |
| Synthesizing, Placing, and Routing a Quick Prototype | 970 |
| Tips for Reducing Runtime | 972 |
| Generating a Best-Performance Prototype | 973 |
| Exploring Different Partitioning Solutions | 977 |
| Scripting Design Tasks | 979 |

CHAPTER 1

Starting with FPGA-Based Prototyping

Short time-to-market windows and the high development costs associated with SoC designs make system validation a high priority for ASIC development teams. FPGA-based prototyping has proven to be an effective method to validate the combination of hardware and software prior to test silicon.

This chapter contains an overview of FPGA-based prototyping and introduces the Synopsys[®] HAPS[®] ProtoCompiler product, which can be combined with the HAPS systems for a hardware-software prototyping solution.

See the following for more information:

- [Introduction to FPGA-Based Prototyping](#), on page 16
- [About the HAPS Prototyping Solution](#), on page 16
- [Starting the Tool](#), on page 27

Introduction to FPGA-Based Prototyping

Market pressures have made it essential to move away from the traditional sequential development of hardware and software, towards a more efficient process where hardware and software development overlap and run in parallel. In this scenario, ASIC and SoC designers require prototypes as soon as possible to validate the hardware-software integration. FPGA prototyping offers a high-performance solution that supports the development of software before silicon becomes available.

FPGA-based prototypes are fully functional hardware representations of an IP block, subsystem or even a complete SoC. FPGA-based prototyping is best used with relatively mature RTL, with much of it fixed or partly verified. The key advantage that FPGA prototypes offer over other prototyping methods is their ability to run at high speed while keeping the RTL implementation as close to the ASIC original as possible. Verification approaches, like simulators, and emulators, are better suited to early debug and functional verification of new design blocks.

- [About the HAPS Prototyping Solution](#), on page 16
- [HAPS ProtoCompiler and ProtoCompiler DX Functionality](#), on page 17
- [HAPS ProtoCompiler Design Flows](#), on page 19
- [HAPS ProtoCompiler DX Design Flows](#), on page 22
- [HAPS Prototyping and Standard Prototyping Use Models](#), on page 24

About the HAPS Prototyping Solution

The Synopsys HAPS Prototyping software provides a complementary development and debug environment for HAPS systems, so that you can prototype multi-FPGA designs. HAPS ProtoCompiler features are designed to improve the productivity of engineers using the HAPS series of FPGA-based prototyping hardware.

The Synopsys HAPS ProtoCompiler DX tool is a powerful combination of hardware and software, and provides a comprehensive and streamlined solution for prototyping single-FPGA designs which target the Xilinx Virtex-7 FPGA family.

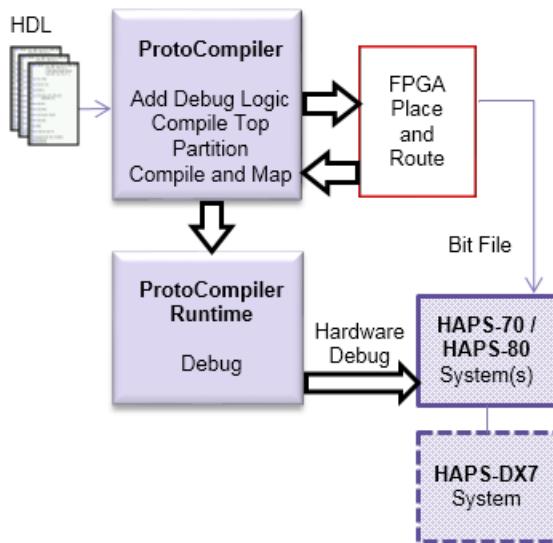
When bundled with the hardware and sold as a complete solution, the product is called HAPS ProtoCompiler S, but the functionality is the same.

HAPS ProtoCompiler and ProtoCompiler DX Functionality

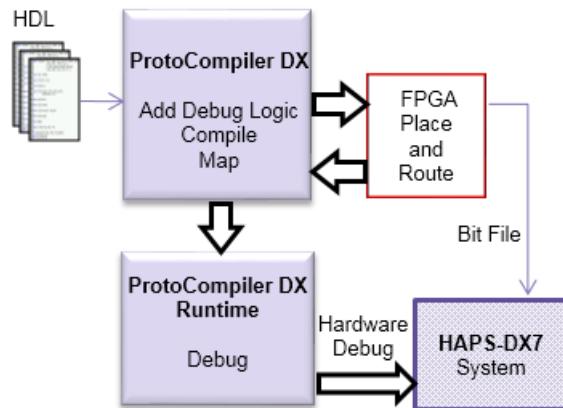
The HAPS ProtoCompiler solution incorporates the following functionality.

- (ProtoCompiler Only) Synopsys HAPS ProtoCompiler prototyping software, which facilitates partitioning across multiple FPGAs, rapid bring-up with fast compilers, and features that simplify migration from ASIC designs and the import of IP.
- (ProtoCompiler DX Only) Synopsys HAPS ProtoCompiler DX prototyping executable, which facilitates rapid bring-up with fast compilers and features that simplify migration from ASIC designs and the import of IP.
- (ProtoCompiler Only) Synopsys HAPS ProtoCompiler runtime, a separate executable, which provides debug functionality.
- (ProtoCompiler DX Only) Synopsys HAPS ProtoCompiler DX runtime, a separate executable, which provides debug functionality.
- (ProtoCompiler Only) Synopsys HAPS-70 Series board systems, with Xilinx Virtex-7 interfaces and I/O interfaces that are compatible with industry-standard FPGA Mezzanine Card (FMC) and HAPS HapsTrak® 3 formats. The HAPS-DX system can be added to function as a daughter board for debug memory.
- (ProtoCompiler DX Only) Synopsys HAPS-DX board system, which includes I/O interfaces that are compatible with industry-standard FPGA Mezzanine Card (FMC) and HAPS HapsTrak® 3 formats.
- (ProtoCompiler Only) Advanced automatic partitioning technology.
- Sophisticated FPGA logic synthesis engine.
- Built-in instrumentation and RTL debug engine.
- Integrated Xilinx place-and-route launch and run.
- Native Universal Multi-Resource Bus (UMRBus®) hardware and C, C++, and Tcl Application Programming Interfaces (APIs).

The following figure summarizes the components that make up the HAPS ProtoCompiler prototyping solution. The optional HAPS-DX7 system is included to illustrate its use as memory for deep trace debug.



The following figure summarizes the various pieces that make up the HAPS ProtoCompiler prototyping solution.



In addition, the software works with other tools to further enhance functionality and provide a seamless prototyping solution:

- Synopsys DesignWare® IP
 - Synopsys VCS simulation
 - Synopsys Verdi™ automated debug

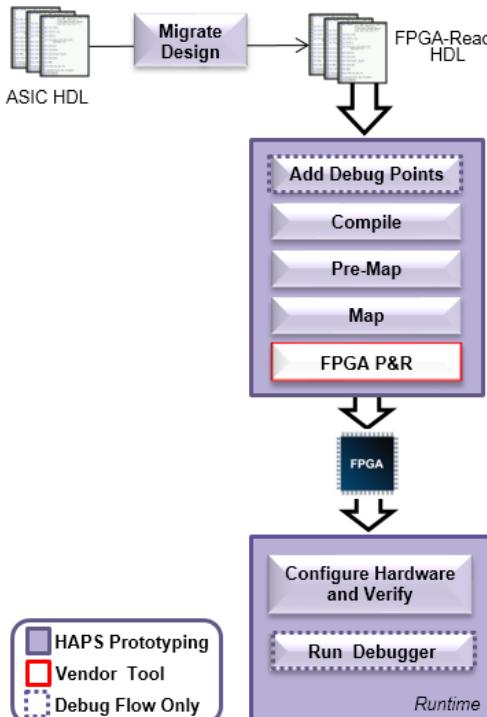
HAPS ProtoCompiler Design Flows

You can use the HAPS ProtoCompiler software with its two component executables to prototype a single-FPGA or a multi-FPGA design. For both, you can include instrumentation and hardware-based debug as part of the flow. The following figures summarize the various design flows available.

Single-FPGA Design Flows

For single-FPGA designs, use the standard synthesis implementation flow or the debug flow. The following figure summarizes the steps in the basic implementation flow (without debug), as well as the debug flow.

The compile and map steps shown are part of the protocompiler executable, hardware configuration uses the Confpro utility (which can be launched from protocompiler_runtime, and the debugger is part of the protocompiler_runtime executable. The hardware consists of HAPS systems with an optional chained HAPS-DX7 system for memory if you are running deep trace debug.

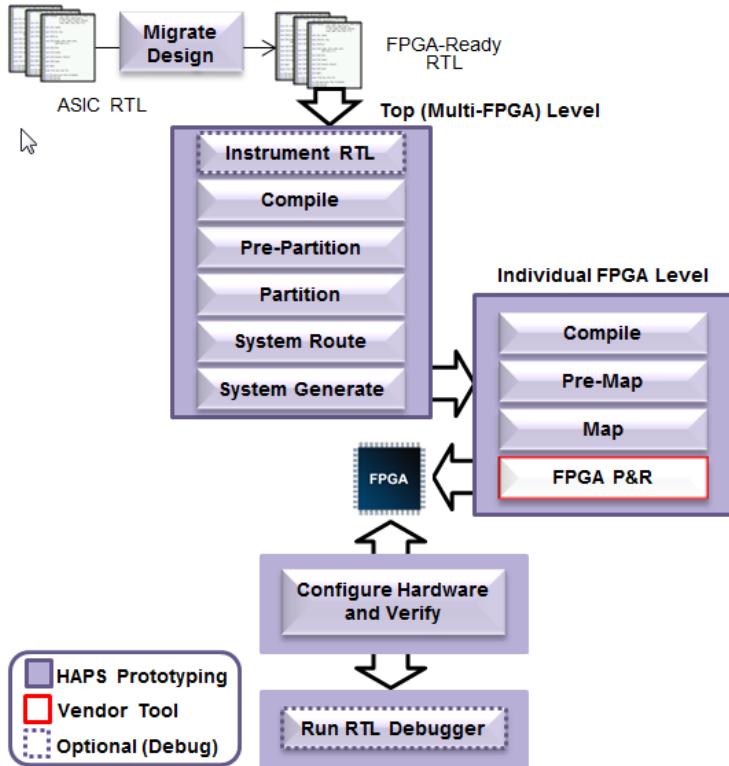


To get details about the steps in the flows, refer to this table:

| | |
|--------------------------------|--|
| Migrating the design from ASIC | Converting ASIC Designs to FPGA, on page 33 |
| Starting the tool | Starting the Tool , on page 27 |
| Setting up and compiling | Creating and Compiling Databases, on page 57 |
| Synthesizing (pre-map and map) | Running the Implementation Flow, on page 471 |
| Debug flow | Instrumenting and Debugging Designs, on page 633 |

Multi-FPGA Design Flows

Multi-FPGA designs require an extra partitioning step in addition to the conversion, synthesis, place and route, and hardware configuration steps that are part of the single-FPGA design flow. As with single-FPGA designs, you can run a basic implementation flow or include hardware-based debug.



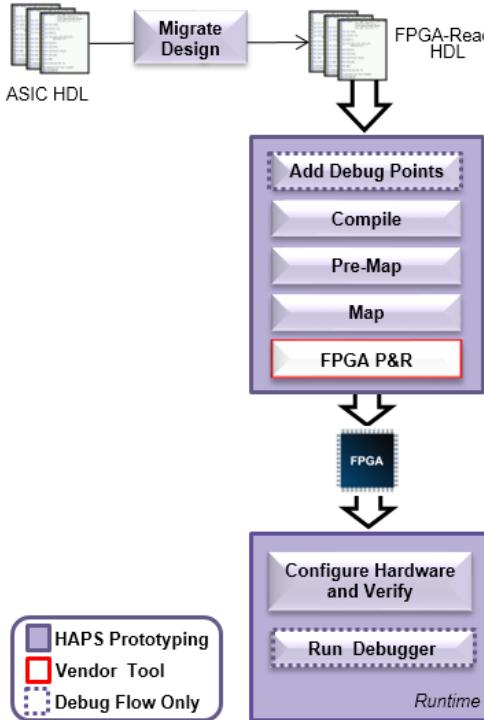
The figure summarizes the steps in the multi-FPGA flow. The implementation steps shown for the top, multi-FPGA level and the individual FPGA level are part of the protocompiler executable, the hardware configuration uses the Confpro utility (which can be launched from `protocompiler_runtime`), and the debugger is part of the `protocompiler_runtime` executable. The hardware consists of HAPS system(s) with an optional chained HAPS-DX7 system for memory if you are running deep trace debug. See [HAPS Prototyping and Standard Prototyping Use Models, on page 24](#) for a diagram of the functionality.

To get details about each of the stages, refer to this table:

| | |
|--|---|
| Migrating the design from ASIC | Converting ASIC Designs to FPGA , on page 33 |
| Starting the tool | Starting the Tool , on page 27 |
| Setting up and compiling | Creating and Compiling Databases , on page 57 |
| Partitioning (includes pre-partitioning, system routing and system generation) | Partitioning the Design , on page 177 |
| Synthesizing (pre-map and map) | Running the Implementation Flow , on page 471 |
| Debug flow | Instrumenting and Debugging Designs , on page 633 |

HAPS ProtoCompiler DX Design Flows

Use the HAPS ProtoCompiler DX software with its two component executables to prototype a single-FPGA design, optionally including hardware-based debug as part of the flow. The following figure summarizes the basic implementation flow, and shows the additional steps needed to implement the debug flow.



The compile and map steps shown are part of the protocompiler_dx executable, the hardware configuration uses the Confpro utility (which can be launched from protocompiler_dx_runtime), and the debugger is part of the protocompiler_dx_runtime executable. The hardware is a HAPS-DX7 system.

To get details about the steps in the flows, refer to this table:

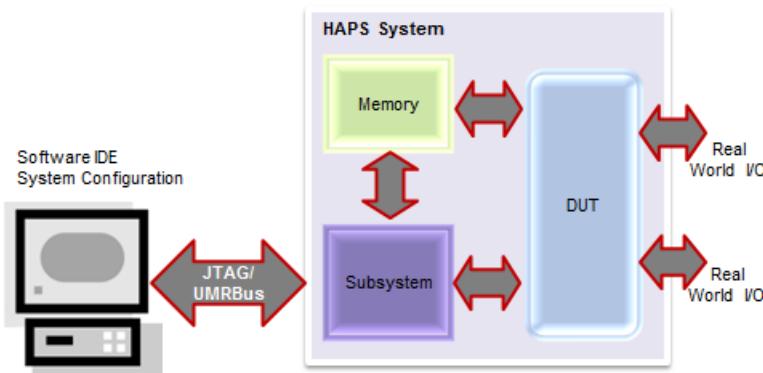
| | |
|-------------------------------------|---|
| Migrating the design from ASIC | Converting ASIC Designs to FPGA , on page 33 |
| Starting the tool | Starting the Tool , on page 27 |
| Setting up and compiling | Creating and Compiling Databases , on page 57 |
| Synthesizing (basic implementation) | Running the Implementation Flow , on page 471 |
| Debug flow | Instrumenting and Debugging Designs , on page 633 |

HAPS Prototyping and Standard Prototyping Use Models

You can plug the HAPS prototyping solution into three prototyping methodologies that are used today:

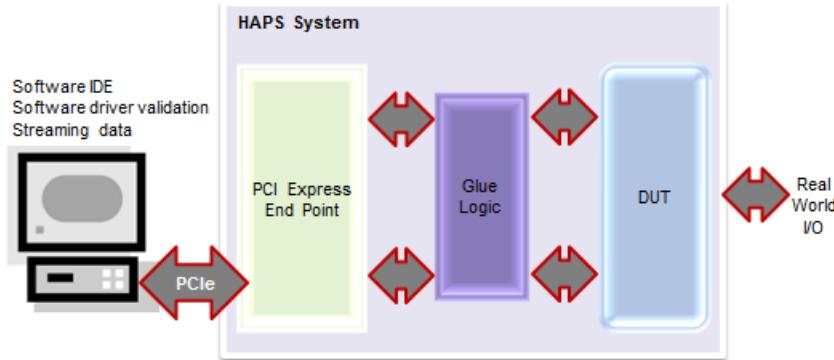
- [Standalone Prototyping Use Model, on page 24](#)
- [Prototyping Use Model with a PCI Express Connection, on page 25](#)
- [Hybrid Prototyping Use Model, on page 25](#)

Standalone Prototyping Use Model



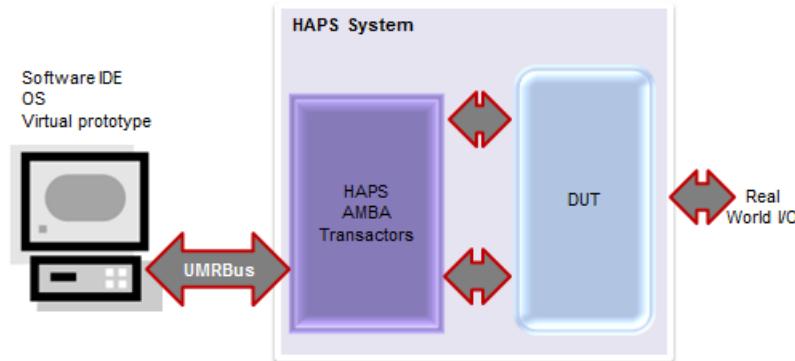
This is the most popular application for FPGA-based prototypes because it is a self-contained environment with real-time clock performance and connects the design under test (DUT) to real world interfaces. It is common practice to use an embedded CPU subsystem of the prototype design to test and execute the software stack. FMC or HapsTrak 3 daughter boards provide PHY interfaces for a wide variety of protocols and systems and the HAPS hardware provides a JTAG connection to external workstations running the software Integrated Development Environment (IDE).

Prototyping Use Model with a PCI Express Connection



In this model, a prototype system is directly plugged into the PCIe slot of a host workstation to enable high-volume data streaming to a DUT. This makes it a popular validation scenario for media controllers. The HAPS Universal Multi-Resource Bus (UMRBus) provides the API to use custom applications to communicate and control the DUT logic. DUT connections are made within the RTL with a simple client interface module.

Hybrid Prototyping Use Model



This model lets you mix SystemC/transaction-level models (TLMs) with FPGA-based prototype hardware. It eliminates RTL availability as a gating factor for system bring-up, and makes prototypes available sooner. The DUT RTL is validated in the context of a virtual processor subsystem running a

software stack of OS and application software. A virtual prototype communicates to the DUT using bus protocol transactors to bridge loosely-timed models with cycle-accurate hardware.

Starting the Tool

The HAPS ProtoCompiler software download includes:

| | |
|---|---|
| protocompiler | Use to synthesize, place, and route |
| protocompiler_runtime | Use for RTL debugging and board system configuration |
| HAPS System Configuration Software (Confpro) and UMRBus Drivers | Includes utilities to program and interface to the HAPS system. |

The HAPS ProtoCompiler DX software download includes:

| | |
|---|---|
| protocompiler_dx | Use to synthesize, place, and route |
| protocompiler_dx_runtime | Use for RTL debugging and board system configuration |
| HAPS System Configuration Software (Confpro) and UMRBus Drivers | Includes utilities to program and interface to the HAPS system. |

See the following for details:

- [Launching the Software](#), on page 27
- [Setting up Additional Tools for the Prototyping Flow](#), on page 30
- [Setting Environment Variables](#), on page 30

Launching the Software

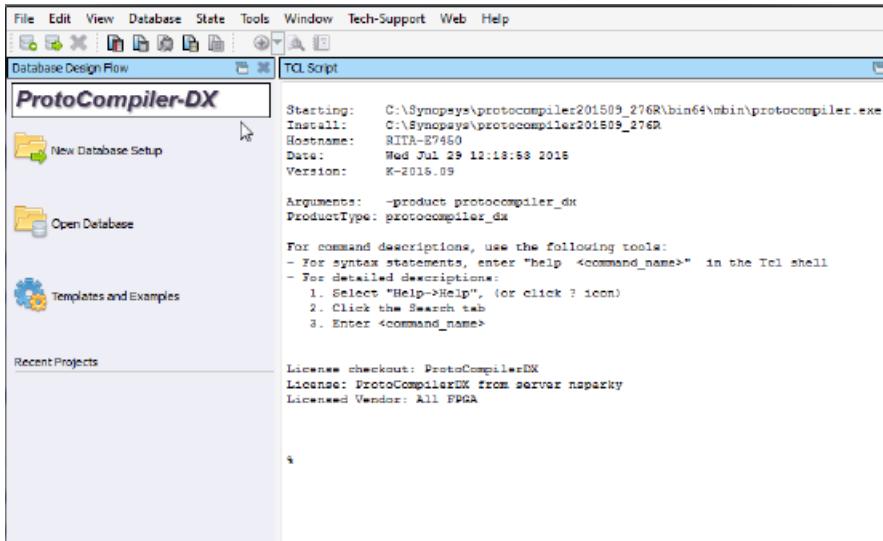
To launch the software, follow these steps:

1. Download and install the software according to the instructions.
You also require the Xilinx Vivado software to place and route, and system configuration software to work with the board system. See [Setting up Additional Tools for the Prototyping Flow](#), on page 30 for details.
2. Set the LM_LICENSE_FILE environment variable to point to your license server, and the PATH variable to include the path to the tool executables.
For information about variables for other functionality used during the prototyping process, see [Setting Environment Variables](#), on page 30.

3. To start the tool, type the appropriate command at a shell prompt:

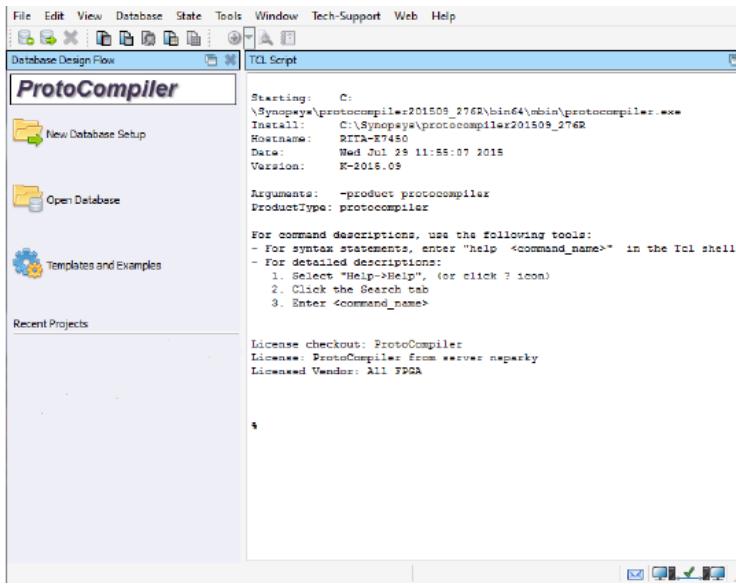
- For ProtoCompiler DX: protocompiler_dx &

The tool console window opens.



- For ProtoCompiler: protocompiler &

The tool console window opens.



4. (ProtoCompilerDX Only) Start implementing your design.

You can run the basic implementation flow or the implementation flow with RTL debug. See [HAPS ProtoCompiler DX Design Flows, on page 22](#) for an overview of the design flow.

For information about the next steps, see [Chapter 2, Converting ASIC Designs to FPGA](#), [Chapter 3, Creating and Compiling Databases](#), or [Chapter 6, Instrumenting and Debugging Designs](#).

5. (ProtoCompiler Only) Determine your working mode, according to what you intend to do:

- Multi-FPGA partitioning

Specify this mode after launching the tool, using the option set `design_flow partition` command. See [Chapter 4, Partitioning the Design](#) for details. This mode is intended for partitioning your large design into multiple FPGAs. You can also instrument your design.

- Single-FPGA implementation mode

Specify this mode after launching the tool, using the option set `design_flow synthesis` command.

This mode is intended for implementing the design at the individual FPGA level, by going through various synthesis, placement, and routing stages. You can also instrument and debug your design.

At the single-FPGA level you can run the basic implementation flow or the implementation flow with RTL debugging. See

[Chapter 2, Converting ASIC Designs to FPGA](#), [Chapter 3, Creating and Compiling Databases](#) or [Chapter 6, Instrumenting and Debugging Designs](#) for the next steps.

See [HAPS Prototyping and Standard Prototyping Use Models](#), on page 24 for an overview of the design flow.

Setting up Additional Tools for the Prototyping Flow

The HAPS ProtoCompiler software lets you implement the prototype design from RTL through synthesis. To proceed through place and route and verification with a board system, there are other tools you must download and install, as shown in the following table. This is not an exhaustive list of tools you could use; for example you might use VCS for simulation.

| Tool | Installation & Documentation Packages |
|---|--|
| HAPS Hardware Documentation | |
| Documentation (<i>Hardware Reference Manual</i>) for the hardware systems (for example HAP-80 S104) for constraint planning | SolvNet > HAPS SupportNet |
| HAPS-DX7 System Documentation | |
| <i>System Hardware Reference Manual</i> for the HAPS-DX7 board system for constraint planning | SolvNet > HAPS SupportNet |
| Xilinx Vivado Design Suite | |
| Includes FPGA place-and-route and bitstream generation functionality required to program the FPGA device | http://www.xilinx.com/ products/design-tools/vivado |

Setting Environment Variables

Set environment variables with the `setenv` command, and provide the path to the functionality.

The following table lists typical environment variables you set for functionality used during prototyping:

| Variable | Description |
|-----------------------------------|--|
| LM_LICENSE_FILE | License server for the prototyping tool |
| PATH | Include path to the prototyping software |
| CDPL_FPGA_HOST | Distributed processing with common distributed processing library (CDPL) |
| SYNOPSYS | Design Compiler® installation |
| XILINX_VIVADO | Place and route with Vivado |
| XILINX | Place and route with ISE |
| VCS_HOME | VCS simulation |
| (ProtoCompiler Only) FORMALITY | Formality equivalence checking |
| HAPS_INSTALL_DIR | Confpro and system configuration installation |

CHAPTER 2

Converting ASIC Designs to FPGA

ASIC or SoC designs are usually much larger and faster than FPGA designs. Additionally, there are various architectural differences between the two, making ASIC designs ready for FPGA prototyping is not a straight-forward process.

This chapter is an overview of the functionality that the Synopsys prototyping tool provides to make it easy to automate or simplify ASIC-to-FPGA translation for prototypes.

- [Getting ASIC Designs Ready for FPGA](#), on page 34
- [Dividing up the Design](#), on page 36
- [Handling I/O Pads](#), on page 37
- [Handling User-Defined Primitives](#), on page 37
- [Converting ASIC Memories](#), on page 38
- [Making ASIC Netlists Suitable for FPGA Design](#), on page 41
- [Converting ASIC Clocks](#), on page 42
- [Dealing with IP](#), on page 53
- [Working with Power Specifications](#), on page 54
- [Converting Constraints](#), on page 55
- [Checking Resources](#), on page 55

Getting ASIC Designs Ready for FPGA

Before an ASIC SoC design can be mapped to an FPGA, some parts of the design must be replaced or reworked because of architectural differences between FPGAs and ASICs, and FPGA capacity limitations.

- ASIC gated clock and generated clock structures do not fit on the FPGA, and must be reworked. Internal fixed clock sources and PLLs must be replaced.
- ASIC memories cannot be used on the FPGA.
 - Replace simple memories with FPGA equivalents generated with dedicated tools from FPGA vendors, or with the Synopsys Symphony Model Compiler product or SYNCORE (included with the Synopsys FPGA synthesis tools).
 - For more complex memories, write replacement models for the FPGA.
- Latches and asynchronous delays must be reworked or replaced.
- ASIC analog modules require wrappers before they can be used on the FPGA.
- I/O control logic for dedicated board systems must be adjusted.

The following sections broadly describe some of the issues:

- [General Guidelines for Moving ASIC Designs to FPGA](#), on page 35
- [Converting ASIC Memories](#), on page 38
- [Converting ASIC Clocks](#), on page 42
- [Dealing with IP](#), on page 53

For a more exhaustive list of conversion issues, see [Converting ASIC Designs to FPGA](#), on page 33.

General Guidelines for Moving ASIC Designs to FPGA

The following guidelines describe some of the best practices to follow with ASIC designs that are to be prototyped in FPGAs:

- Identify and deal with FPGA-hostile RTL. The following ASIC structures do not easily map to FPGA architectures. See the rest of the chapter for more information about handling these conversions.
 - Asynchronous delays in latches need to be removed, reworked or replaced with clocked processes.
 - ASIC memory is too big and must be split across multiple RAMs
 - Clocks with gating. Gated clocks usually overflow the FPGA clock resources. See *Converting ASIC Clocks, on page 42* for details.
 - Complex generated clocks need to be simplified to fit the FPGA resources. See *Converting ASIC Clocks, on page 42* for details.
 - Clock muxing, as for embedded test logic
 - Top-level pads. See *Handling I/O Pads, on page 37* for details.
 - Gate-level netlists
 - Analog modules require wrappers to interface to external circuitry.
 - User-defined primitives (UDP) that specify the behavior of an element or module. If you include them in the design you get an error during logic synthesis. Replace them with equivalent RTL that describes the function of the UDP.
 - SoC leaf cells instantiations.
 - SoC memories.
 - SoC-specific IP without source RTL.
 - BIST and other test circuitry instantiated in the RTL. Handle them by leaving these signals dangling.
- Avoid latches because they are hard to time on an FPGA.
- Avoid combinatorial loops.
- To ensure that the ASIC RTL is portable to FPGA, do not include optimizations like clock gating, test insertion, and low-power in the original RTL. Instead, leave optimizations to the SoC tools and keep the RTL pure as far as possible.

- Use ‘define’ and ‘ifdef’ to include or remove code for prototyping. Use these constructs to isolate BIST (built-in self test), memory instantiations, etc.
- Isolate RTL changes to within the library elements rather than outside them in the RTL structure. This improves portability and keeps the prototyping code as close to the original as possible.
- Share make files between SoCs and FPGAs by using macro-driven branching for different targets.
- Reduce the impact of changes to the ASIC source RTL, by using wrappers to make changes. Replace files instead of editing them, and backannotate all changes.

Dividing up the Design

ASIC or SoC designs are much larger than FPGA designs, and are usually divided up into partitions across multiple FPGAs before prototyping at the individual FPGA level.

The HAPS ProtoCompiler tool is hardware-aware and includes various tools to automate the partitioning process as much as possible. See [Chapter 4, Partitioning the Design](#) for details.

Handling I/O Pads

There are two ways to handle the ASIC I/O pad ring:

- Prototype just the ASIC design core, ignoring the I/O pad ring. The FPGA synthesis engine can infer and insert I/O buffers, so you can leave out the I/O pad ring and make the dangling connections inactive or tie them to the top-level boundary.
- Keep the ASIC I/O pad ring in the FPGA prototype design.
 - Create an FPGA-synthesizable equivalent for every ASIC I/O pad by writing the equivalent RTL or instantiating it. The FPGA equivalent only needs to model the logical connection from the design core to the outside world. You need to create one for each different kind of I/O pad.
 - Replace each ASIC I/O pad with the corresponding synthesizable model you created.

Handling User-Defined Primitives

The FPGA logic synthesis tool does not synthesize user-defined primitives (UDPs). If you include them for FPGA synthesis, you get the following error:

```
@E:CG731 : test.v(3) | Can't synthesize UDP primitives
```

Replace any UDPs in your design with the equivalent RTL function.

Converting ASIC Memories

ASIC memory is typically much larger and more complex than FPGA memory. FPGA memory is typically implemented within the FPGA, while ASIC memory could be off-chip.

1. Convert ASIC memory as appropriate to your design needs.

See [Guidelines for Converting ASIC Memory](#), on page 39.

2. Run logic synthesis.

The software automatically infers RAMs from the RTL code and reports them. The tool can infer and implement FPGA RAM from register banks in the RTL. You can also guide inference by specifying the `syn_ramstyle` attribute before synthesizing the design. For detailed information on inferring RAM, refer to [Chapter 3, Defining Objects for Inference](#) in the *Compiler Mapper Guide*.

3. Check the results.

- Check the usage summary in the log file to see if the FPGA memory resources are over-utilized:

```
RAM/ROM usage summary
Block Rams: 160 of 156 (102%)
Register bits not including I/Os: 2500 (50%)
```

- Check the log file for a report of the inferred memories. For example:

```
#### START OF Block RAM DETAILED REPORT ####
Total Block RAMs: 1
mem_mem_0_0
-----
READ_WIDTH_A    4
WRITE_WIDTH_A   4
READ_WIDTH_B    4
WRITE_WIDTH_B   4
WRITE_MODE_A    READ_FIRST
SRVAL_A         0
-----
#### END OF Block RAM DETAILED REPORT ####
```

Guidelines for Converting ASIC Memory

Here are some recommended guidelines for converting ASIC memory:

- Remove ASIC memory that is not needed for the prototype, and implement the FPGA with external RAM.
- Replace ASIC memory with equivalent FPGA models whenever possible.
- You can instantiate the memories or write them as HDL code from which the software can infer them, and then map them to the appropriate resources.
 - You can create different wrappers for the memories in the HDL code, one for ASIC and one for FPGA. The FPGA wrapper either describes the behavior so that the memory can be inferred and mapped correctly, or has a vendor-provided netlist describing the behavior.
 - Use the `syn_ramstyle` attribute to specify the kind of RAMs to infer. See [Setting Attributes to Guide RAM Inference, on page 209](#) in the *Compiler Mapper Guide* for details.
 - For latch-based RAMs, specify latch ram inference before compiling the design (option set `latchram 1` command). The compiler schematic shows the latch RAM inferred:



You can then replace the inferred `latchram1` memories with FPGA equivalents. Make sure to do so before mapping; you get an error if there are `latchram1` instances when you map the design.

- Disable test and power pins on ASIC memory, by ignoring them or tying them to constants.
- Make sure to thoroughly test the functionality of FPGA-compatible memory models with testbenches before using them in your design. Some tools, like SYNCORE, provide testbenches.

Techniques for Large ASIC Memories

- Map complex memories to FPGA models if possible.
- If the complex memory is too large to fit on the FPGA, implement it using off-chip resources external to the FPGA, like a DDR3 or DDR4. Connect and synchronize the external memory to the FPGA. Black-box the memory on the FPGA.
- Create two RTL wrappers, one for the FPGA prototype and one for ASIC. The FPGA wrapper should have RTL that describes the behavior so that the FPGA synthesis tool can infer the memory structure and map it to the appropriate vendor technology.

Techniques for Smaller ASIC Memories

- Directly replace less complex memories with FPGA-specific memories generated by vendor tools.
- Use the netlists provided by FPGA vendors for certain memories.

Making ASIC Netlists Suitable for FPGA Design

One of the biggest challenges is to translate the original golden ASIC netlist to make it suitable for FPGA design. If you edit the original ASIC RTL netlist you do not have to modify the HDL source for small changes to the netlist. Editing the netlist lets you keep the HDL code intact. Post-synthesis netlist editing is useful for implementing engineering change orders (ECOs), disabling ASIC clocks, rerouting a fast clock from an external source to an internal (DCM) source, and inserting or “stitching” IP blocks in an RTL netlist (`srs` file) at a desired level of hierarchy.

For further information about netlist editing techniques, see [Editing Netlists, on page 907](#), which includes information about these topics:

- [Editing Netlists with HAPS ProtoCompiler Commands](#), on page 907
- [Editing a Verilog Netlist](#), on page 910
- [Using Cross-Module Referencing \(XMR\)](#), on page 913
- [Using Gate-Level ASIC Netlists](#), on page 911
- [Creating Parallel Code with force and bind Statements](#), on page 911

Converting ASIC Clocks

ASIC clocks are typically quite complex, with numerous clocks that are “balanced” using clock tree synthesis to avoid clock skew. Unlike ASICs, FPGAs have a finite number of low-skew global clocks that are built in as part of the architecture. In addition, ASIC clocks might be dynamically programmed by software, and have asynchronous clock domains, or domains that are gated for power saving. If you have a gated clock in an FPGA implementation, it adds extra delay and upsets the balance, introducing timing violations and other glitches in the process.

ASIC clocks must be converted to fit the restricted resources of FPGA architecture. The prototyping tool includes features like gated clock conversion (GCC) feature to automate the bulk clock conversion, but it requires that the clock information be properly defined. You must also specify decisions about trade-offs to the tool.

For details, see the following:

- [General Guidelines for ASIC Clocks](#), on page 42
- [Defining Clocks](#), on page 44
- [Scaling External Clock Speed](#), on page 47
- [Converting Gated Clocks](#), on page 862

General Guidelines for ASIC Clocks

The following guidelines describe some ways to handle the differences between ASIC and FPGA clocking schemes.

Simplifying the ASIC Clock Network

- Avoid scan or test logic on the clock path. For prototyping purposes, tie scan logic to 0, so that the tool automatically removes this logic when it runs. Test logic on the clock path prevents gated clock conversion, which results in high clock skew and hardware instability.
- If it is necessary that the Scan Clock and the associated mux logic on the clock path be retained, replace the logic mux with the FPGA BUFGMUX primitive. This ensures that the clock is routed on the dedicated clock network, this and reduces the clock skew caused by local routing.

- Swap the ASIC PLLs with functionally equivalent MMCM structures on the FPGA. Use the MMCMs to generate the subsystem, interface, IP, and peripheral clocks required for the design. The reference clocks for these MMCMs must come from the HAPS system PLLs whenever feasible. If the reference clock for the user FPGA PLL/MMCM has to be an external hardware clock source, you can further redistribute the clock back onto the HAPS global clock network.

Clock Mapping Guidelines

- Map design clocks to HAPS sources. Clocks that are not mapped can cross FPGAs, and can therefore cause skew.
- Map gated clocks to the HAPS sources or replicate the clocks as needed. Make sure to map system-level clocks generated within the FPGAs to the GCLK network.
- Avoid one-to-one mapping of cascaded dividers on clock sources when moving ASIC/SoC code to the FPGA. Instead, merge cascaded circuits and use the new divide value to program the MMCM output.
- Do not manually instantiate BUFG and IBUFGDS on clock nets and ports. The tool inserts them automatically, according to the loads on the clocks.

Synchronizing Design Resets and MMCM Lock Signals

- The lock times of different MMCMs are not guaranteed to occur at the same time. It is recommended that you poll the lock signals of all the MMCMs in the design as well as the locked signals, and then use this result to create a delayed power on master reset for the entire design.
- It is recommended that you use a separate reset to control the MMCMs. This reset must be different from the master reset discussed above, and is used to decouple the clock reset circuit from the design reset.

Replicating Clock Networks

- To avoid problems implementing the design in the hardware, replicate clock networks in these cases:
 - When you exhaust the HAPS clocks and need to use FPGA MMCMs to generate clocks, replicate clocks in each FPGA.

- If you have source-synchronous clocks that are generated by the FPGA, replicate them in the FPGAs, because mapping source-synchronous clocks to HAPS clocks makes them asynchronous.
- Do not replicate MMCMs that generate odd multiples of the source clock across FPGAs. If such clocks are needed across systems, map them to the HAPS GCLKs, with the connecting FPGA as the source. MMCMs that create even multiples of the source clock across FPGAs (2^n) can be replicated.
- Do not replicate MMCMs that have divide-by clocks across FPGA boundaries.
- Monitor the status of the lock signals of replicated MMCMs across systems, and use the result to control the master reset for the design.

Defining Clocks

- [Defining Clocks using HAPS ProtoCompiler](#), on page 44
- [Defining Clocks using HAPS ProtoCompiler DX](#), on page 46

Defining Clocks using HAPS ProtoCompiler

For ProtoCompiler, you must map the design clocks to the HAPS system clock infrastructure, because the HAPS global clock (GCLK) network is used to distribute clocks between the FPGAs and generate clocks. In addition, specify timing constraints for the design clocks so that the tool correctly recognizes, converts, and maps them during the synthesis process.

The following figure shows where the clocks are defined and the different kinds of information defined for the clocks. The multi-FPGA partitioned flow shows the clock definition files superimposed on the HAPS hardware resources and the steps in the prototyping flow, thus illustrating the kind of information defined and the place in the flow where the definitions are used. Clock definitions for the single-FPGA partitioning or hybrid flow follow the multi-FPGA model. Clock definitions for the single-FPGA model exclude information about the HAPS system configuration.

1. Start with a compiled design and specify HAPS-aware definitions for the clocks.
 - Define port bins for the FPGAs with the `board_system_configure -top_io` command.

```
board_system_configure -top_io{FB1.A4}
board_system_configure -top_io{FB1.B1}
```

- Identify the HAPS backbone or global clocks and define the clock sources for these GCLKs in the TSS file, using `board_system_configure -clock` commands. See [Defining Clocks in the TSS File, on page 194](#) for details.
 - Specify how global clocks connect to design clocks with `net_attribute` and `assign_global_net` constraints in the PCF file. See [Assigning Clocks in the PCF, on page 251](#) for more information.
2. Partition the design.
- Use the TSS and PCF information for partitioning.
 - Replicate clocks as needed to handle clock skew across partitions. See [Partitioning Clocks, on page 348](#) for more information.
3. Set timing constraints for the design clocks in the FDC file.
- These constraints set up the tool to recognize the design clocks and correctly map them to the FPGA resources.
- Declare the top-level master clocks and their frequencies with `create_clock` constraints. Clocks that are not declared can be inferred by the tool. Avoid inferred clocks, because the tool uses default settings for them and does not apply many optimizations to them.
 - Set constraints to define asynchronous clock domains using the `set_clock_group` constraint. Clocks within the same clock group are synchronous with each other.
 - Set constraints for clock muxing.
- See [Specifying FDC Constraints, on page 20](#) in the *Compiler-Mapper Guide*.
4. Set constraints for gated clocks and generated clocks in the FDC file.
- Declare gated clocks and generated clocks (clocks derived from the master clock) with `create_generated_clock` constraints.
 - Enable gated clock conversion (GCC).
- See [Converting Gated Clocks, on page 862](#) for information about gated clocks, generated clocks, and GCC.
5. After mapping, check results.

See [Checking Clock Information, on page 546](#) for an overview of clock information checks available at different stages of the design.

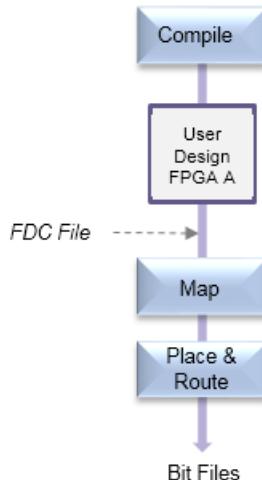
6. Place and route the design.
7. Configure the frequencies for the clock generators.
 - Export the TSS information with the `export runtime` command. This command creates a project file for Confpro with all the clock source information correctly configured.
 - Use Confpro to configure the frequency for the clock generators and to associate the clock generators with the global clocks.

Note that the GCLK you use determines which clock generator you use to set the frequency. For example, with GCLK1, you must use PLL1 to set the frequency.

Defining Clocks using HAPS ProtoCompiler DX

For ProtoCompiler DX, you must specify timing constraints for the design clocks so that the tool correctly recognizes, converts, and maps them during the synthesis process.

The following figure shows the file where the clocks are defined in the context of the single-FPGA prototyping flow in HAPS ProtoCompiler DX tool.



1. After compiling, set timing constraints for the design clocks in the FDC file.

These constraints set up the tool to recognize the design clocks and correctly map them to the FPGA resources.

- Declare the top-level master clocks and their frequencies with `create_clock` constraints. Clocks that are not declared can be inferred by the tool. Avoid inferred clocks, because the tool uses default settings for them and does not apply many optimizations to them.
- Set constraints to define asynchronous clock domains using the `set_clock_group` constraint. Clocks within the same clock group are synchronous with each other.
- Set constraints for clock muxing.

See [Specifying FDC Constraints, on page 20](#) in the *Compiler-Mapper Guide*.

2. Set constraints for gated clocks and generated clocks in the FDC file.

- Declare gated clocks and generated clocks (clocks derived from the master clock) with `create_generated_clock` constraints.
- Enable gated clock conversion (GCC).

See [Converting Gated Clocks, on page 862](#) for information about gated clocks, generated clocks, and GCC.

3. After mapping, check results.

See [Checking Clock Information, on page 546](#) for an overview of clock information checks available at different stages of the design.

4. Place and route the design.

Scaling External Clock Speed

Use the clock scaling feature to tune the external clocks to the fastest clock speed on a HAPS-80 system, and the design executes as expected (realizes the timing requirements). The idea is to find the fastest speeds of clocks in a design, that still allows all delay paths to meet the timing requirements (all slacks are zero or positive).

The Tcl option to enable the clock scaling:

```
option set clock_scaling <multi_root>
```

Clock scaling is invoked at two points during the ProtoCompiler flow:

- Before the timing report is written out during system generate stage.
 - This mode uses estimated timing data.
- ```
report timing -generate -out slta
```
- Before the timing report is written out during system-level timing analysis (SLTA).
    - This mode uses the back-annotated data from tool's place-and-route run.

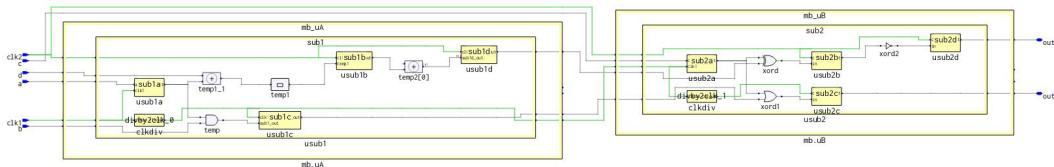
```
report timing -generate -mode slta -out slta
```

## Analyze the Clock Scaling Report

The clock frequency scaling report is generated under the corresponding clock Performance Summary table, in the regular timing report. You need to observe the values in the Estimated Frequency or Estimated Period columns.

This section shows an example of the output in the timing report, with and without clock scaling, in system generate stage with estimated values and BA SLTA. Without clock scaling, all the clocks are scaled independently whereas when clock scaling is enabled, all the external clocks that are synchronous are scaled using the same ratio and the asynchronous clocks are scaled independently.

## Schematic



## Asynchronous Clocks

### FDC constraints

```
create_clock -name {clk1} [get_ports {p:clk1}] -period 40.0
create_clock -name {clk2} [get_ports {p:clk2}] -period 30.0
create_generated_clock -name {genclk1} -source {p:clk1} [get_pins
{usub1.clkdiv.clkout}] -divide_by 2
create_generated_clock -name {genclk2} -source {p:clk1} [get_pins
{usub2.clkdiv.clkout}] -divide_by 4
set_clock_groups -asynchronous -group {clk1 genclk1 genclk2}
set_clock_groups -asynchronous -group {clk2}
```

### Performance Summary in the System Generate Stage

Without clock scaling:

Here, the root clocks clk1 and clk2, and the generated clocks genclk1 and genclk2 are scaled independently.

@S0.0 | Performance Summary  
\*\*\*\*\*

| Starting Clock  | Requested Frequency | Estimated Frequency | Requested Period | Estimated Period | Slack   | Clock Type                                     | Clock Group        |
|-----------------|---------------------|---------------------|------------------|------------------|---------|------------------------------------------------|--------------------|
| HAPS_lock_clkin | 1.0 MHz             | NA                  | 1000.000         | NA               | NA      | declared (HAPS_lock_clkin)                     | haps_lock_clkgroup |
| HAPS_umr_clk    | 100.0 MHz           | 1221.9 MHz          | 10.000           | 0.818            | 9.182   | derived (from HAPS_umr_clkin) (HAPS_umr_clkin) | default_clkgroup   |
| HAPS_umr_clkin  | 100.0 MHz           | NA                  | 10.000           | NA               | DCM/PLL | declared (HAPS_umr_clkin)                      | default_clkgroup   |
| clk1            | 25.0 MHz            | 1438.2 MHz          | 40.000           | 0.695            | 39.305  | declared                                       | group_1_107        |
| clk2            | 33.3 MHz            | 186.9 MHz           | 30.000           | 5.351            | 24.649  | declared                                       | group_1_108        |
| genclk1         | 12.5 MHz            | 719.1 MHz           | 80.000           | 1.391            | 79.413  | generated (from clk1)                          | group_1_107        |
| genclk2         | 6.2 MHz             | 107.0 MHz           | 160.000          | 9.346            | 150.654 | generated (from clk1)                          | group_1_107        |
| System          | 1.0 MHz             | 312.8 MHz           | 1000.000         | 3.196            | 996.803 | system                                         | system_clkgroup    |

With clock scaling:

Here, the asynchronous clocks clk1 and clk2 are scaled independently, but the generated clocks genclk1 and genclk2 are scaled with respect to the root clock clk1. Scaling rate for clk1 is 0.1 and for clk2 is 0.18.

@S0.0 | Performance Summary  
\*\*\*\*\*

| Starting Clock  | Requested Frequency | Estimated Frequency | Requested Period | Estimated Period | Slack   | Clock Type                                     | Clock Group        |
|-----------------|---------------------|---------------------|------------------|------------------|---------|------------------------------------------------|--------------------|
| HAPS_lock_clkin | 1.0 MHz             | NA                  | 1000.000         | NA               | NA      | declared (HAPS_lock_clkin)                     | haps_lock_clkgroup |
| HAPS_umr_clk    | 100.0 MHz           | NA                  | 10.000           | NA               | 9.182   | derived (from HAPS_umr_clkin) (HAPS_umr_clkin) | default_clkgroup   |
| HAPS_umr_clkin  | 100.0 MHz           | NA                  | 10.000           | NA               | DCM/PLL | declared (HAPS_umr_clkin)                      | default_clkgroup   |
| clk1            | 25.0 MHz            | 250.0 MHz           | 40.000           | 4.000            | 39.305  | declared                                       | group_1_107        |
| clk2            | 33.3 MHz            | 185.2 MHz           | 30.000           | 5.400            | 24.649  | declared                                       | group_1_108        |
| genclk1         | 12.5 MHz            | 125.0 MHz           | 80.000           | 8.000            | 79.413  | generated (from clk1)                          | group_1_107        |
| genclk2         | 6.2 MHz             | 62.5 MHz            | 160.000          | 16.000           | 150.654 | generated (from clk1)                          | group_1_107        |
| System          | 1.0 MHz             | NA                  | 1000.000         | NA               | 996.803 | system                                         | system_clkgroup    |

### Performance Summary in the SLTA

## Without clock scaling:

Here, the root clocks clk1 and clk2, and the generated clocks genclk1 and genclk2 are scaled independently.

| Starting Clock                                                                                                                   | Requested Frequency | Estimated Frequency | Requested Period | Estimated Period | Slack   | Clock Type  |        |
|----------------------------------------------------------------------------------------------------------------------------------|---------------------|---------------------|------------------|------------------|---------|-------------|--------|
| HAPS0_0   Performance Summary                                                                                                    |                     |                     |                  |                  |         |             |        |
| Worst slack in design: 8.282                                                                                                     |                     |                     |                  |                  |         |             |        |
|                                                                                                                                  |                     |                     |                  |                  |         |             |        |
| Starting Clock                                                                                                                   | Requested Frequency | Estimated Frequency | Requested Period | Estimated Period | Slack   | Clock Type  |        |
| HAPS0_0   SystemIP_1a_0_11a_440a_2016_06_1_1_mb_uh_sipgen I_haps00_systemip_core_I_unr_clk_gen_mmcu_clkout2_derived_clock_CLKIN1 | 100.0 MHz           | NA                  | 10,000           | NA               | NA      | derived (:) |        |
| HAPS0_0   SystemIP_1a_0_11a_440a_2016_06_1_1_mb_uh_sipgen I_haps00_systemip_core_F_unr_clk_gen_mmcu_clkout2_derived_clock_CLKIN1 | 100.0 MHz           | NA                  | 10,000           | NA               | NA      | derived (:) |        |
| HAPS_clk_12_5                                                                                                                    | 12.5 MHz            | NA                  | 80,000           | NA               | NA      | derived (:) |        |
| HAPS_clk_200                                                                                                                     | 200.0 MHz           | NA                  | 5,000            | NA               | NA      | derived (:) |        |
| HAPS_clk_100K                                                                                                                    | NA                  | NA                  | 100,000,000      | NA               | NA      | declared    |        |
| HAPS_1mz_clk                                                                                                                     | 100.0 MHz           | 582.0 MHz           | 10,000           | 1,718            | 8,282   | derived (:) |        |
| HAPS_10mz_clk                                                                                                                    | 100.0 MHz           | NA                  | 10,000           | NA               | DCM/PLL | declared    |        |
| clk1                                                                                                                             | 25.0 MHz            | 138.4 MHz           | 40,000           | 5,233            | 34.747  | declared    |        |
| clk2                                                                                                                             | 33.3 MHz            | 72.3 MHz            | 30,000           | 11,702           | 18.222  | declared    |        |
| genclk1                                                                                                                          | 6.2 MHz             | NA                  | 80,000           | NA               | 72.222  | generated   |        |
| genclk2                                                                                                                          | 6.2 MHz             | 35.3 MHz            | 160,000          | 28,186           | 131.834 | generated   |        |
| System                                                                                                                           | 1.0 MHz             | NA                  | 1000,000         | NA               | 8.254   | 991.746     | system |

## With clock scaling:

Here, the asynchronous clocks clk1 and clk2 are scaled independently, but the generated clocks genclk1 and genclk2 are scaled with respect to the root clock clk1. Scaling rate for clk1 is 0.18 and for clk2 is 0.46.

| Starting Clock                                                                                                                   | Requested Frequency | Estimated Frequency | Requested Period | Estimated Period | Slack   | Clock Type  |          |
|----------------------------------------------------------------------------------------------------------------------------------|---------------------|---------------------|------------------|------------------|---------|-------------|----------|
| HAPS0_0   Performance Summary                                                                                                    |                     |                     |                  |                  |         |             |          |
| Worst slack in design: 8.282                                                                                                     |                     |                     |                  |                  |         |             |          |
|                                                                                                                                  |                     |                     |                  |                  |         |             |          |
| Starting Clock                                                                                                                   | Requested Frequency | Estimated Frequency | Requested Period | Estimated Period | Slack   | Clock Type  |          |
| HAPS0_0   SystemIP_1a_0_11a_440a_2016_06_1_1_mb_uh_sipgen I_haps00_systemip_core_I_unr_clk_gen_mmcu_clkout2_derived_clock_CLKIN1 | 100.0 MHz           | NA                  | 10,000           | NA               | NA      | derived (:) |          |
| HAPS0_0   SystemIP_1a_0_11a_440a_2016_06_1_1_mb_uh_sipgen I_haps00_systemip_core_F_unr_clk_gen_mmcu_clkout2_derived_clock_CLKIN1 | 100.0 MHz           | NA                  | 10,000           | NA               | NA      | derived (:) |          |
| HAPS_clk_12_5                                                                                                                    | 12.5 MHz            | NA                  | 80,000           | NA               | NA      | derived (:) |          |
| HAPS_clk_200                                                                                                                     | 200.0 MHz           | NA                  | 5,000            | NA               | NA      | derived (:) |          |
| HAPS_100K_clk                                                                                                                    | NA                  | NA                  | 100,000,000      | NA               | NA      | declared    |          |
| HAPS_1mz_clk                                                                                                                     | 100.0 MHz           | NA                  | 10,000           | NA               | 8,282   | derived (:) |          |
| clk1                                                                                                                             | 25.0 MHz            | 10.0 MHz            | 40,000           | NA               | NA      | DCM/PLL     | declared |
| clk2                                                                                                                             | 33.3 MHz            | 15.0 MHz            | 30,000           | 11,800           | 18.222  | declared    |          |
| genclk1                                                                                                                          | 6.2 MHz             | NA                  | 80,000           | NA               | 72.222  | generated   |          |
| genclk2                                                                                                                          | 6.2 MHz             | NA                  | 160,000          | 28,186           | 131.834 | generated   |          |
| System                                                                                                                           | 1.0 MHz             | NA                  | 1000,000         | NA               | 991.746 | system      |          |

## Synchronous Clocks

### FDC Constraints:

```
create_clock -name {clk1} [get_ports {p:clk1}] -period 40.0
create_clock -name {clk2} [get_ports {p:clk2}] -period 30.0
create_generated_clock -name {genclk1} -source {p:clk1} [get_pins
{usub1.clkdiv.clkout}] -divide_by 2
create_generated_clock -name {genclk2} -source {p:clk1} [get_pins
{usub2.clkdiv.clkout}] -divide_by 4
```

## Performance Summary in the System Generate Stage

### Without clock scaling:

Here, the root clocks clk1 and clk2, and the generated clocks genclk1 and genclk2 are scaled independently.

```
#S0.0 |Performance Summary

```

Worst slack in design: 3.851

| Starting Clock     | Requested Frequency | Estimated Frequency | Requested Period | Estimated Period | Slack   | Clock Type                                             | Clock Group        |
|--------------------|---------------------|---------------------|------------------|------------------|---------|--------------------------------------------------------|--------------------|
| HAPS_lock_clkinkin | 1.0 MHz             | NA                  | 1000.000         | NA               | NA      | declared (HAPS_lock_clkinkin)                          | haps_lock_clkgroup |
| HAPS_urnr_clk      | 100.0 MHz           | 1221.9 MHz          | 10.000           | 0.818            | 9.182   | derived (from HAPS_urnr_clkinkin) (HAPS_urnr_clkinkin) | default_clkgroup   |
| HAPS_urnr_clkinkin | 100.0 MHz           | NA                  | 10.000           | NA               | DCM/PLL | declared (HAPS_urnr_clkinkin)                          | default_clkgroup   |
| clk1               | 25.0 MHz            | 415.5 MHz           | 40.000           | 2.407            | 9.398   | declared                                               | default_clkgroup   |
| clk2               | 33.3 MHz            | 54.2 MHz            | 30.000           | 18.447           | 24.649  | declared                                               | default_clkgroup   |
| genclk1            | 12.5 MHz            | 20.3 MHz            | 80.000           | 49.192           | 3.851   | generated (from clk1)                                  | default_clkgroup   |
| genclk2            | 6.2 MHz             | 107.0 MHz           | 160.000          | 9.346            | 150.654 | generated (from clk1)                                  | default_clkgroup   |
| System             | 1.0 MHz             | 312.8 MHz           | 1000.000         | 3.196            | 996.803 | system                                                 | system_clkgroup    |

With clock scaling:

Here, the synchronous clocks clk1 and clk2 are scaled to the same ratio. The generated clocks genclk1 and genclk2 are scaled with respect to the root clock clk1. Scaling rate for clk1 and clk2 is 0.62.

```
#S0.0 |Performance Summary

```

Worst slack in design: 3.851

| Starting Clock     | Requested Frequency | Estimated Frequency | Requested Period | Estimated Period | Slack   | Clock Type                                             | Clock Group        |
|--------------------|---------------------|---------------------|------------------|------------------|---------|--------------------------------------------------------|--------------------|
| HAPS_lock_clkinkin | 1.0 MHz             | NA                  | 1000.000         | NA               | NA      | declared (HAPS_lock_clkinkin)                          | haps_lock_clkgroup |
| HAPS_urnr_clk      | 100.0 MHz           | NA                  | 10.000           | NA               | 9.182   | derived (from HAPS_urnr_clkinkin) (HAPS_urnr_clkinkin) | default_clkgroup   |
| HAPS_urnr_clkinkin | 100.0 MHz           | NA                  | 10.000           | NA               | DCM/PLL | declared (HAPS_urnr_clkinkin)                          | default_clkgroup   |
| clk1               | 25.0 MHz            | 40.3 MHz            | 40.000           | 24.800           | 9.398   | declared                                               | default_clkgroup   |
| clk2               | 33.3 MHz            | 53.8 MHz            | 30.000           | 18.660           | 24.649  | declared                                               | default_clkgroup   |
| genclk1            | 12.5 MHz            | 20.2 MHz            | 80.000           | 49.600           | 3.851   | generated (from clk1)                                  | default_clkgroup   |
| genclk2            | 6.2 MHz             | 10.1 MHz            | 160.000          | 99.200           | 150.654 | generated (from clk1)                                  | default_clkgroup   |
| System             | 1.0 MHz             | NA                  | 1000.000         | NA               | 996.803 | system                                                 | system_clkgroup    |

## Performance Summary in the SLTA

Without clock scaling:

Here, the root clocks clk1 and clk2, and the generated clocks genclk1 and genclk2 are scaled independently.

```
#S0.0 |Performance Summary

```

Worst slack in design: -7.807

| Starting Clock                                                                                                          | Requested Frequency | Estimated Frequency | Requested Period | Estimated Period | Slack   | Clock Type  |
|-------------------------------------------------------------------------------------------------------------------------|---------------------|---------------------|------------------|------------------|---------|-------------|
| HAPS8888EMIP_is_0_is_4468_2016_86_1_mb_uA_alpgen[1].haps#0_systemip_core_1_urn_clk_gen_mmcn_clkout2_derived_clock_CLKR1 | 100.0 MHz           | NA                  | 10.000           | NA               | NA      | derived (f) |
| HAPS8888EMIP_is_0_is_4468_2016_86_1_mb_uA_alpgen[1].haps#0_systemip_core_1_urn_clk_gen_mmcn_clkout2_derived_clock_CLKR1 | 100.0 MHz           | NA                  | 10.000           | NA               | NA      | derived (f) |
| HAPS_clk_12_5                                                                                                           | NA                  | NA                  | NA               | NA               | NA      | derived (f) |
| HAPS_clk_200                                                                                                            | NA                  | NA                  | NA               | NA               | NA      | derived (f) |
| HAPS_urnr_clkinkin                                                                                                      | NA                  | NA                  | NA               | NA               | NA      | derived (f) |
| HAPS_urnr_clk                                                                                                           | NA                  | NA                  | NA               | NA               | NA      | derived (f) |
| clk1                                                                                                                    | 25.0 MHz            | 582.0 MHz           | 40.000           | 1.718            | 8.282   | derived (f) |
| clk2                                                                                                                    | 33.3 MHz            | 65.3 MHz            | 30.000           | 10.494           | 7.376   | derived (f) |
| genclk1                                                                                                                 | 12.5 MHz            | 7.0 MHz             | 80.000           | 13.422           | 13.433  | declared    |
| genclk2                                                                                                                 | 6.2 MHz             | 36.3 MHz            | 160.000          | 142.458          | -7.807  | generated   |
| System                                                                                                                  | 1.0 MHz             | 105.1 MHz           | 1000.000         | 9.511            | 996.483 | system      |

Estimated period and frequency reported as NA means no slack depends directly on the clock waveform

With clock scaling:

Here, the synchronous clocks clk1 and clk2 are scaled to the same ratio. The generated clocks genclk1 and genclk2 are scaled with respect to the root clock clk1. Scaling rate for clk1 and clk2 is 1.79.

| 800.0   Performance Summary                                                                                                    |                     |                     |                  |                  |         |                      |
|--------------------------------------------------------------------------------------------------------------------------------|---------------------|---------------------|------------------|------------------|---------|----------------------|
|                                                                                                                                | Requested Frequency | Estimated Frequency | Requested Period | Estimated Period | Slack   | Clock Type           |
| <b>Starting Clock</b>                                                                                                          |                     |                     |                  |                  |         |                      |
| HAPS80_SYSTEMIP_1s_0_11s_440ns_2016_06_1_1_mb_uA_sipgen 1_haps80_systemip_core_1_umi_clk_gen_mmcn_cikout2_derived_clock_CLKIN1 | 100.0 MHz           | NA                  | 10.000           | NA               | NA      | derived (1)          |
| HAPS80_SYSTEMIP_1s_0_11s_440ns_2016_06_1_1_mb_uB_sipgen 1_haps80_systemip_core_1_umi_clk_gen_mmcn_cikout2_derived_clock_CLKIN1 | 100.0 MHz           | NA                  | 10.000           | NA               | NA      | derived (1)          |
| HAPS_clk_12_5                                                                                                                  | 12.5 MHz            | NA                  | 80.000           | NA               | NA      | derived (1)          |
| HAPS_clk_200                                                                                                                   | 200.0 MHz           | NA                  | 5.000            | NA               | NA      | derived (1)          |
| HAPS_lock_cikin                                                                                                                | 1.0 MHz             | NA                  | 1000.000         | NA               | NA      | declared (1)         |
| HAPS_umi_clk                                                                                                                   | 100.0 MHz           | NA                  | 10.000           | NA               | 8.982   | derived (1)          |
| HAPS_umi_cikin                                                                                                                 | 100.0 MHz           | NA                  | 10.000           | NA               | NA      | DCM/PLL declared (1) |
| clk1                                                                                                                           | 25.0 MHz            | 14.0 MHz            | 40.000           | 71.600           | 7.376   | declared             |
| clk2                                                                                                                           | 100.0 MHz           | 100.0 MHz           | 10.000           | 59.200           | 18.796  | declared             |
| genclk1                                                                                                                        | 12.5 MHz            | 9.0 MHz             | 80.000           | 135.200          | -7.807  | generated            |
| genclk2                                                                                                                        | 6.2 MHz             | 3.5 MHz             | 160.000          | 266.400          | 132.477 | generated            |
| System                                                                                                                         | 1.0 MHz             | NA                  | 1000.000         | NA               | 990.493 | system               |

Estimated period and frequency reported as NA means no slack depends directly on the clock waveform

## Limitation

The scaling ratio is based on the inter-fpga critical paths. This is because intra-fpga critical paths are not visible in the SLTA or the SLTA-BA.

# Dealing with IP

ASIC designs can include different types of IP (intellectual property); for example, Synopsys DesignWare® components, interface IP, CPU subsystems, third-party digital IP, and analog IP. For more information about incorporating IP, see [Chapter 6, Working with IP](#) in the *Compiler Mapper Guide*.

There are two main reasons to include IP in the FPGA prototype:

- Re-validation

In this scenario, the impetus is to check the functionality of a particular IP implementation. For example, the team might take the DesignWare IP for a USB 3.0 interface and confirm the functions of the controller configuration, controller-to-PHY operation, PHY electrical compliance, and driver development.

- IP integration

In this scenario, the prototyping solution serves as the platform for integrating multiple IPs or integrating new IP with a processor subsystem. You can use software to confirm peripheral access, measure access latency, and confirm the operation of software algorithms with real-world interfaces connected to the prototype.

# Working with Power Specifications

IEEE 1801-2009 Unified Power Format (UPF) is a standard for power specification. UPF provides a portable low-power design specification for interoperability between various tools. It allows you to focus on power as a key consideration early in the design process by defining power domains and islands.

## ASIC and FPGA Power Domains

ASIC power domains are quite different from FPGA power domains:

| FPGAs                                                                                                                                             | ASICs                                            |
|---------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------|
| Volatile. Lose their configuration contents when they are powered down; the configuration must be reloaded from onboard flash memory on power up. | Do not lose configuration when powered down.     |
| The design cannot be powered down selectively for certain regions.                                                                                | Can power down parts of the design.              |
| No dedicated power switches.                                                                                                                      | Dedicated power switches.                        |
| No multi-voltage or user-configurable cells.                                                                                                      | Power domains can operate at different voltages. |
| Single power and a single ground for mapping.                                                                                                     | Multiple power and ground.                       |

Because of these differences, power domains are implemented as virtual power domains on an FPGA. The device is not powered down, but the tool emulates the behavior of the powered-down state. For details on adapting and using UPF specifications, see [Including UPF Specifications, on page 827](#).

# Converting Constraints

The tool processes some standard formats automatically, which reduces the time taken to translate constraints.

- SDC timing constraints

Migrate to the FDC file for FPGA synthesis as is, because FDC uses the same syntax. If you have clocks that were not converted, use the following find commands to find unconverted clocks for analysis:

```
find -seq -hier {*} -filter {@clock==*clockNamePattern*}
select [find -seq -hier {*} -filter {@clock==*clockNamePattern*}]
```

For information about including timing constraints in the design, refer to [Chapter 5, Running the Implementation Flow](#).

- UPF power constraints

Put UPF constraints into a file and then read them into the prototype design. See [Including UPF Specifications, on page 827](#) for details.

# Checking Resources

It is essential that you check available resources when you map an ASIC to an FPGA.

1. Run synthesis.
  - For first-pass resource estimation, run fast synthesis strategy.
  - For optimized synthesis, synthesize as usual.
2. Check the Resource Usage report for I/Os, random logic, flip-flops, memory, and clocks. Aim for 50% utilization.

```
RAM/ROM usage summary
Block Rams: 160 of 156 (102%)
Register bits not including I/Os: 2500 (50%)
```

3. If the design is over-utilized, use register resources (distributed RAM) to implement the memories.

Use the `syn_ramstyle` attribute to guide memory inference.



## CHAPTER 3

# Creating and Compiling Databases

---

This chapter describes the basics of the prototyping database model and how to compile the database. The compile step is the initial step in an FPGA synthesis or partitioning design flow.

This chapter contains the following information:

- [Working with a Design Database](#), on page 58
- [Compiling the Design](#), on page 77
- [Adding Design Files](#), on page 96
- [Specifying Source Files \(Standard Compiler\)](#), on page 97
- [Using UUM and Group Mapping](#), on page 109
- [Using Unified Compile](#), on page 120
- [Preparing the Input for Unified Compile](#), on page 125
- [Compiling Incrementally](#), on page 144
- [Specifying Constraints](#), on page 167
- [Setting Options](#), on page 171

# Working with a Design Database

Once your design is FPGA-ready, you must set up your design database. A design database consists of all the data files required for a particular design. The database model consists of various database states. The database states represent the design at intermediate transformative stages.

1. Get your ASIC design ready for FPGA synthesis, placement, and routing.
  - See [Chapter 2, Converting ASIC Designs to FPGA](#) for a discussion of SoC-to-FPGA conversion issues.
  - See [Setting Environment Variables, on page 30](#) for environment variable settings.
2. Open the prototyping tool and either create or load a database, with the database create and database load commands, respectively.

For details about working with databases, see the following procedures:

- [Creating a Design Database, on page 58](#)
- [Loading and Navigating Database States, on page 62](#)
- [Converting Legacy Projects to Databases, on page 65](#)
- [Archiving and Unarchiving Databases, on page 66](#)

## Creating a Design Database

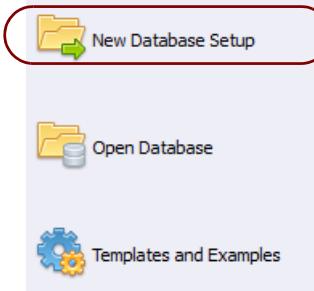
Every design requires a database. The first step in prototyping a design is to set up the design database.

1. Type database create followed by the directory location at the tool prompt.

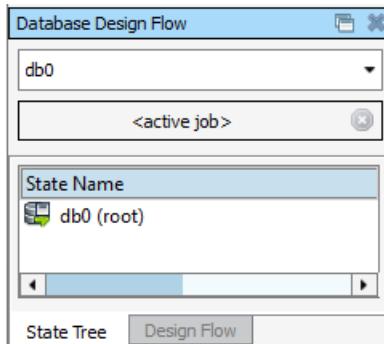
database create *databaseDirectory*

There are various other options you can specify with the database command line; see [database, on page 30](#) in the *Command Reference* manual for the complete command syntax.

To use the GUI to create a database, click New Database Setup in the left panel of the tool UI and specify the database directory when prompted.



By default the tool creates a design database in the current directory. If you do not name the database, the tool names it db0 (root) by default. If you are using the GUI, the directory is displayed in the panel on the lower left of the window, with a green arrow next to it to indicate that it is the active state.



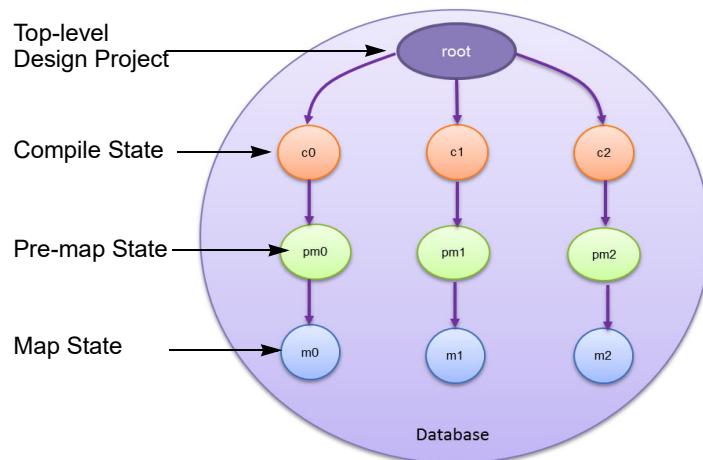
You can now run operations on the root database (State menu in the GUI) to generate database states. A database state is an implementation of the root db0 database at a particular design phase, based on a unique set of design options. The database states serve as points where you can check that you are converging on performance and area requirements, and where you can tweak constraints and options to ensure that you achieve your design goals.

Subsequent commands work on the active database state (db0 in this case), unless you load another database or switch to another state.

The database is a closed database. Direct access and use of the files in the database directory structure is not recommended; instead, use the `export` commands to export the files you need. The structure of the database and file format can change without notice.

The database hierarchy starts at root, and branches through the states at different design stages. The source files you specified apply to root and every state below it, but there might be additional files, like constraint and option files, that you can specify for different database states.

The following figure is a simple illustration of synthesis database states with default names. You can return to any database state and rerun a command with a new set of options; this creates a new branch and another parallel database state. For example, you could return to pm2 and rerun mapping to generate an m3 state under pm2. See [Design Database Structure, on page 11](#) of the *Reference Manual* for a more detailed discussion of databases.



2. Specify the HAPS technology with the `database create -technology` command.

The specified technology becomes the default technology associated with this database and applies to subsequent database states, unless you reset it with the `database apply_state -technology` command at the compile stage. You can have different database states that use different target technologies.

For multi-FPGA designs with a mixture of HAPS systems, set the technology to the largest system. If you have a mixture of HAPS-70 and HAPS-80 systems, set the technology to HAPS-80. The other information is inferred from the TSS file, when you set it up later in the design cycle.

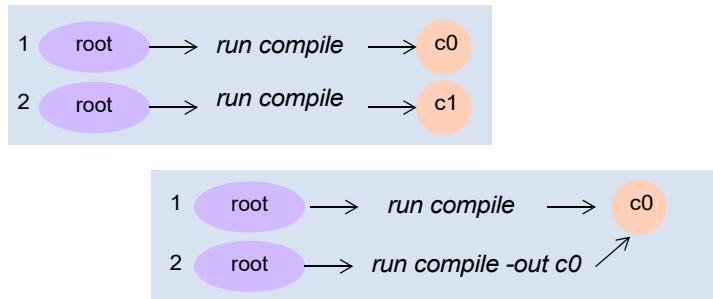
3. If you do not want to use the default database names, you can specify a name:
  - For the root state, specify a name when you create the database, with `database create`. In the GUI, you can specify a different name in the dialog box that opens when you create the database.
  - For any other database state, specify the name with the `-out` option when you use the `run` command that generates that state. For example, specifying `run -compile trial1` generates a database state called `trial1` instead of the default, `c0`.
  - Rename an existing state with the `database rename_state` command. Alternatively right-click the state name in the GUI and select `Rename State`. For example, this command renames `sr10` to `sysroute_best_tdm8`:

```
database rename_state {sysroute_best_tdm8} -state {sr10}
```

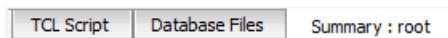
Note that renaming a state does not change the underlying directory; this means that both the old and new names are illegal names for new states until you remove the state.

4. To overwrite an existing database instead of creating a new one, specify the name of the database you want to overwrite, using the `-out` option to the `implementation` command.

The following figure illustrates how the `run compile -out` command overwrites a database state.



5. In the GUI, click the Summary tab at the bottom of the Tcl shell window.



This tab shows details of all database states run for the current database tree.

6. To populate a database, you must specify the source files.

For details about adding files, see [Adding Design Files, on page 96](#).

## Loading and Navigating Database States

Once you load a database state, subsequent commands apply to that database until you close it or reset it to another database state. The GUI panel on the lower left indicates the active database.

The database is a closed database. Direct access and use of the files in the database directory structure is not recommended; instead, use the export commands to export the files you need, or the report commands to view the reports generated after the command is run for that database state. The structure of the database and file format can change without notice.

1. To load a database, do the following:

- If you currently have another design database open, close the active database with the database close command. See [database, on page 30](#) in the *Command Reference* manual for the complete command syntax for the database commands.
- Type this command at the console prompt:

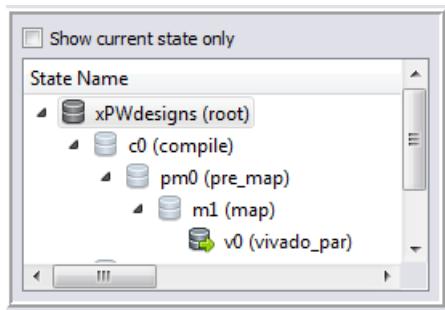
```
database load databaseName
```

The GUI equivalents for this command are the Database->Database Open and Database->Database Create commands. If you are starting a design, you can now specify your source files and design options, as described in [Adding Design Files, on page 96](#).

2. Make sure you are at the right point in the database tree for the implementation commands you want to run.

All states are preserved by default. Implementation commands operate on data in the active database state. Any state can be accessed at any time, so it is important that you are in the appropriate state in the database hierarchy when you run the implementation commands.

From the command line, use the database get command to identify the active database state. In the GUI, the State Name panel at the lower left of the tool window shows the database tree, and indicates the active database with a green arrow. A red database state indicates that the database state has errors.

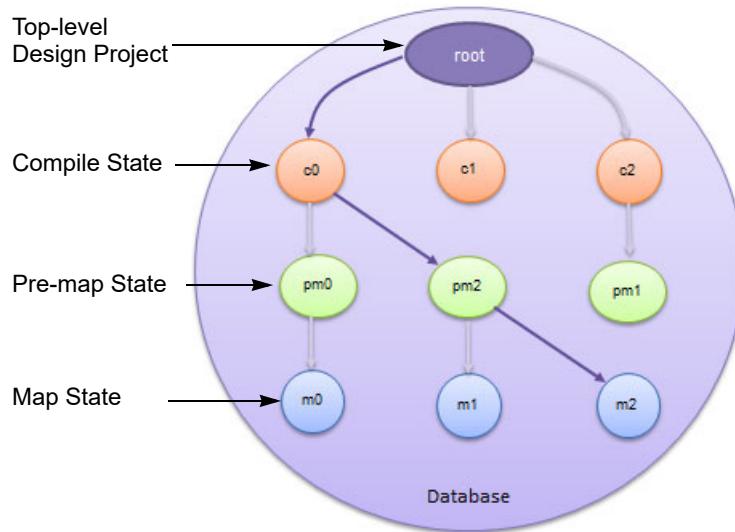


3. Use the database set {*stateName*} command to switch to another database state.

In the GUI, you can double-click on the database state you want in the Database Evolution panel to switch to that state.

You can return to any database state and rerun a command with a new set of options to branch and generate a parallel database state. This modular approach gives you flexibility to explore design options by creating parallel database states and allowing you to select the state with which you want to proceed.

The following example shows different branching database states created at the compile, pre-map, and map phases as options are explored. The database states have the default names and are numbered consecutively as they are generated. The darker arrows indicate the chosen database states at each phase. Premap states were generated from the most promising compile states, c0 and c2, but the premap states generated from c0 (pm0 and pm2) proved better than the pm1 state generated from c2 for moving further with the implementation and generating mapped database states.



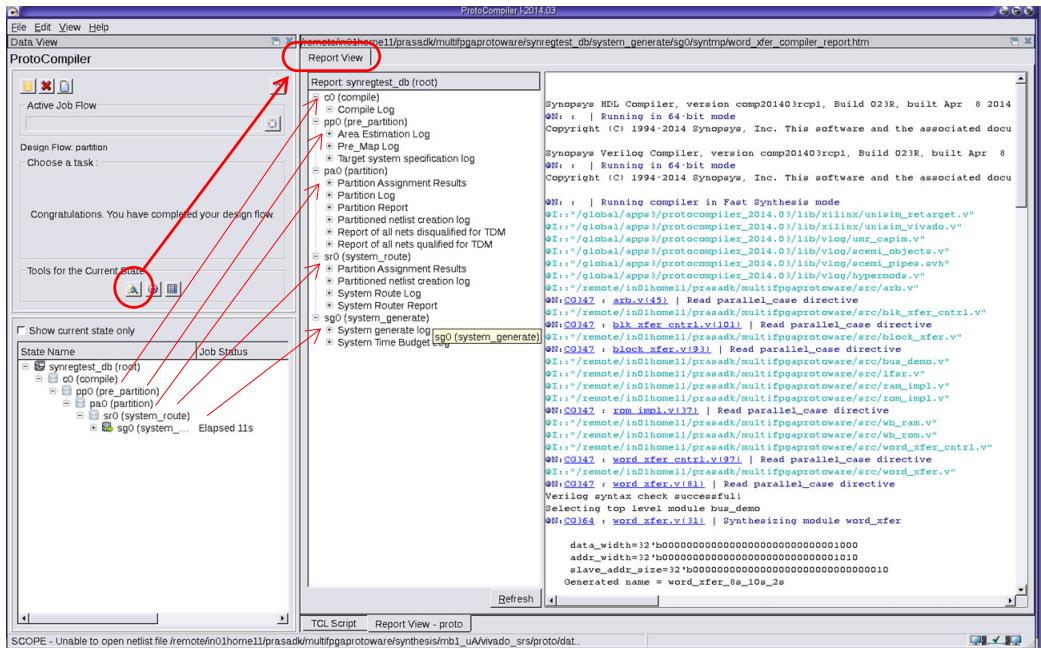
The figure shows that you can return to a previous database state at any point and proceed from there. If you want to place and route m1 instead of m2, you can switch to the m1 database state and then run place and route to generate v0. Similarly, you could return to c2 and run premapping again to generate pm3.

4. To remove a database state that you do not want, use database `remove_state`.

Alternatively, you can overwrite an existing database that you know you do not want to keep by specifying the appropriate `run` command with the `-out` option, listing the name of the existing state you want to overwrite. For example, `run map -out m1` overwrites the existing m1 state, instead of generating m2.

5. You can query a database state, or check the files generated for a database state.
  - For information about querying a database state use the `database query` and `database query_state` commands. For more information about querying, see [Querying Incremental Results, on page 520](#). For more information about the command syntax, use the `help` command. For example, type `help database query_state`.
  - Each database state generates its own reports, appropriate to the `run` command used to generate that database state. Clicking the Report

View icon in the GUI or specifying the export report command displays the reports available for the active stage, and all its ancestors in the database tree. In the following figure, the active state is the System Generate state, but the Report view displays the reports available for all the ancestors of this state.



Reports are part of the closed database. To open and save reports outside the database, export them (export command) or use the methods described in [Checking Reports and Log Files, on page 512](#) to view them in HTML or access them offline.

## Converting Legacy Projects to Databases

If you have a legacy Synplify Premier or Certify project that you want to convert to a database to use with the standard compiler flow, follow these steps:

1. Source this conversion utility:

```
source install_dir/examples/haps_utilities/convert2protocompiler.tcl
```

The utility is a static conversion script that converts the basic synthesis project so that you can compile the design in the prototyping tool. You must add other information manually.

2. Type this command to start conversion:

```
convert2protocompiler prjFilePath outputDirectory
```

The utility uses the Synplify Premier project file at the path specified and sets up an initial database and the design files that the compiler needs to start prototyping. It creates a text file that lists all the source files. It also generates Tcl options and constraints files based on the options and constraints specified for the Synplify Premier project. Finally, it generates a script file you can use to automatically run prototyping on the design.

3. Add other information that the script does not handle.
  - Check that the technology target is set to the appropriate HAPS technology.
  - If you have Certify partitioning files and other project information that the script does not handle, add that information manually.
  - Set the mode you want to work in with the option set command. Set it to either partitioning or the basic synthesis implementation.

## Archiving and Unarchiving Databases

The tool includes an archive utility that lets you store the files for a database into a single archive file, which is in a proprietary (sar) format. You can archive an entire database tree or selected files from the database. You can also restore the files from an archived database. See these links for more information:

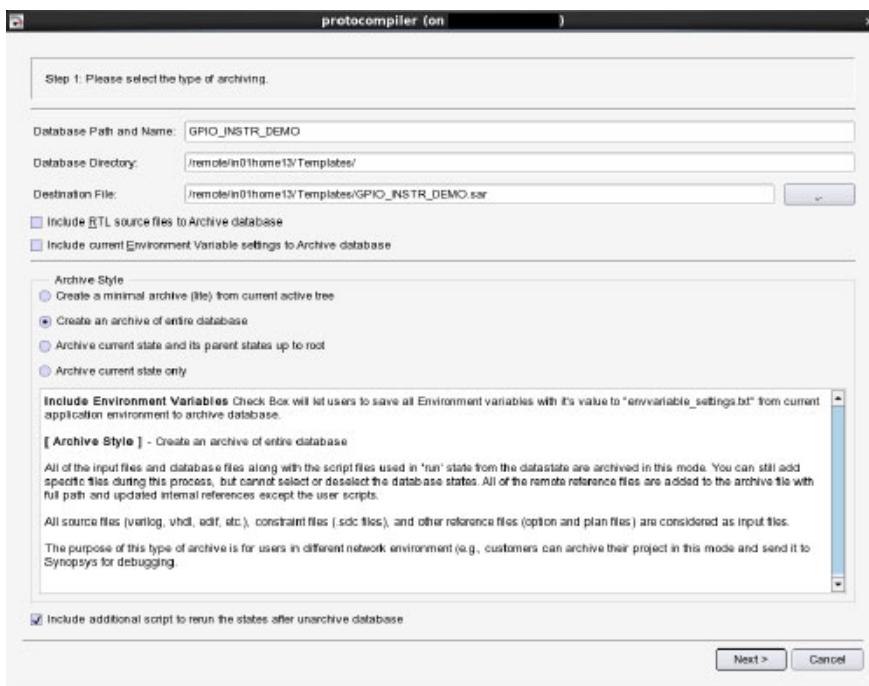
- [Archiving a Database](#), on page 67
- [Extracting Database Files from an Archive](#), on page 71
- [Creating Minimal \(Lite\) Archives](#), on page 74

## Archiving a Database

Either use the database archive command to create an archive file or use the GUI. The latter provides an easy interface to add, delete, and view the files for archiving.

1. In the Data View, select Database -> Archive Database to open the utility.

The Tcl equivalent to the GUI is the database archive *directoryPath* command. See [database archive, on page 32](#) in the *Command Reference Manual* for the syntax.



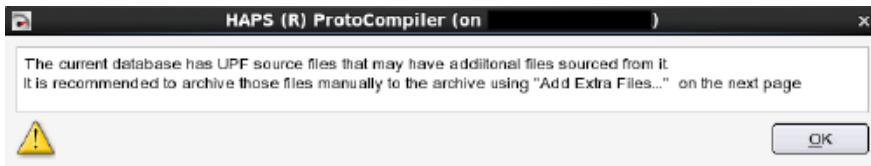
2. Do the following on the Step 1 screen:
  - Enter a location for the archive file in Destination File.
  - To include the source RTL files in the archive, select Include RTL Source (database archive -include\_source 1). If the box is not selected the tool archives all user input files, except for RTL files.
  - To include all Environment variables with values in the archive database, select the Include Current Environment Variable settings

to Archive database. All Environment variables and values are saved to the *envvariable\_settings.txt* file in the archive database.

- Select the Archive Style. The equivalent command line option is -mode. You can archive the entire database (database archive -mode full), the current tree (database archive -mode active\_tree), or the current state ((database archive -mode current). Active tree mode archives the current database state and its parent states back to root, and the current state mode only archives the current database state. The default is active tree (current state and its ancestors up to the root). For information about creating a minimal archive, see [Creating Minimal \(Lite\) Archives, on page 74](#).
- Note that if you only archive the current state, you cannot use any run commands on the archived state later. You can only generate reports and schematics. To be able to use run commands to generate downstream database states, archive the active tree (-mode active\_tree) or the entire database (-mode full).

The archive utility archives just the database and state information files. It does not archive reports or other files. If the current tree contains a pre-map or pre-partition state, the tool archives all the input needed to re-run this state.

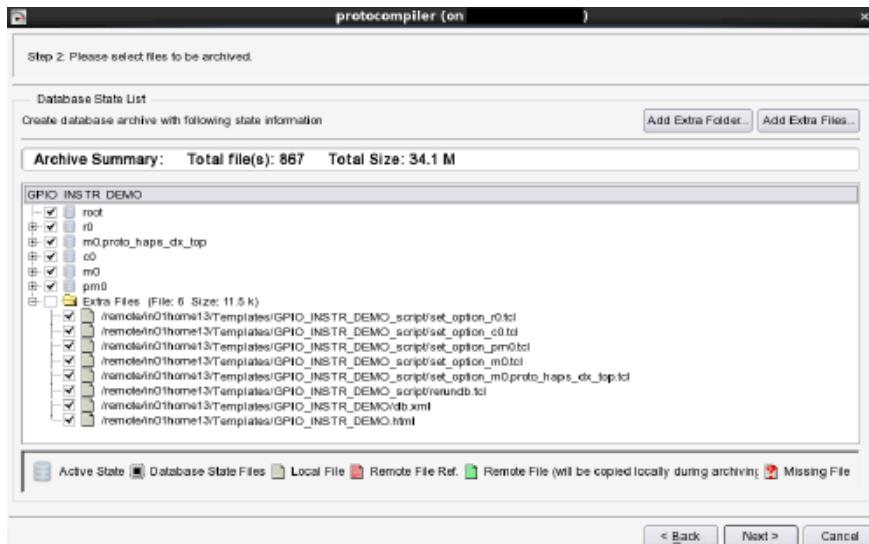
If you have included UPF files to the list of files to be archived, the following message is displayed:



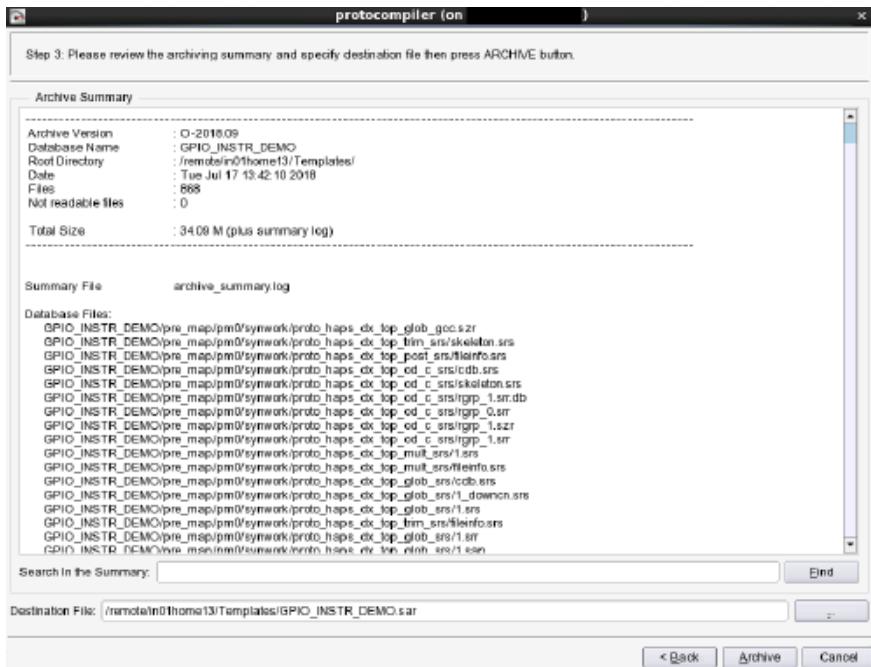
For information about using run commands with a minimal archive (-mode lite), see [Creating Minimal \(Lite\) Archives, on page 74](#).

- Click Next. The utility displays the Step 2 summary of the files to be included in the archive and shows the full uncompressed file size for the archive.
3. In the Step 2 screen, deselect the files you do not want to archive.
- Double click the directory categories and use the check boxes to add or delete files.

- You can add additional script files to the archive database using the Add Extra Folder or Add Extra Files buttons. To add individual script files to the archive database, use the Add Extra Files button. To add a directory with multiple script files in it, use the Add Extra Folder button. These files will be archived and unarchived at the same level as the database.
- Click Next when you are done. This action advances you to the Step 3 confirmation screen.



4. On the next screen, verify the destination and other information.

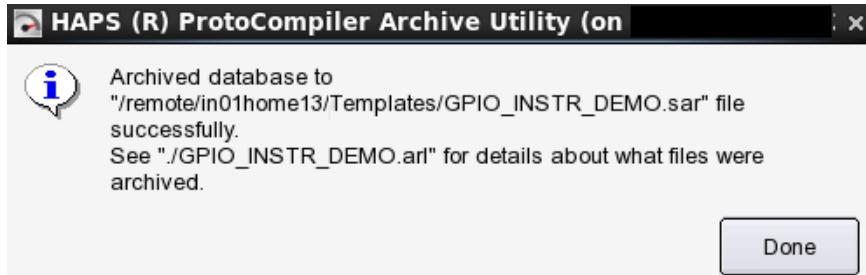


- Check that the archive contains the file set that you want to archive and that destination archive filename is correct.
- If the list of files is not correct, click the Back button and include/exclude the files as needed.
- Click Archive when you are satisfied.

The archive utility automatically runs the report syntax\_check command on the database to ensure that the files are syntactically correct. It archives the entire database, including source files. If your design has Verilog 'include' files, the utility includes the complete list of Verilog files. If you specified that option, it archives the database states from the current state to the root. The utility displays the name of the project to archive, the top-level directory where the project file is located (root directory), and other information.

## 5. Check the archive utility report.

The archive utility reports the file generation success and the path location of the archive file.



6. Click Done to close the dialog box.

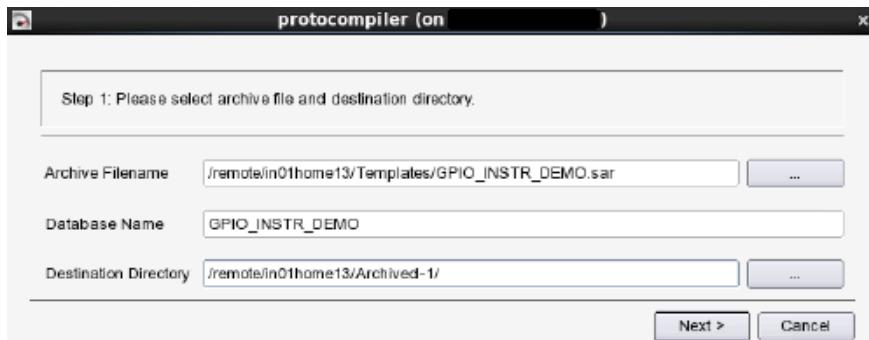
## Extracting Database Files from an Archive

Use the following procedure to extract the design database files from a sar archive file, or *unarchive* the database. You can also use the database unarchive Tcl command to extract the files instead of the GUI method described here.

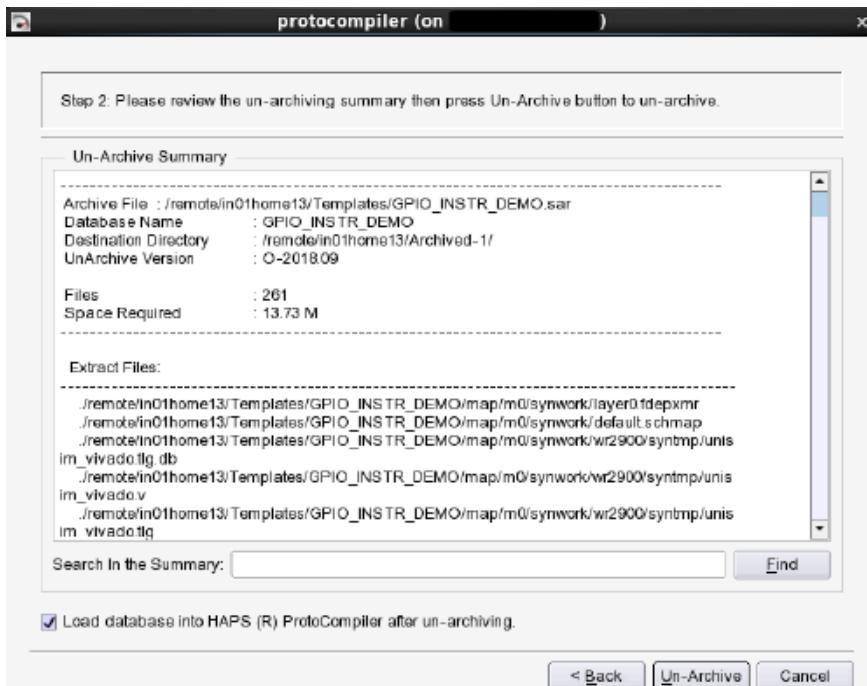
1. From the tool window, select Database->Un-Archive Database to display the utility.

The Tcl command equivalent is database unarchive *filename.sar*. See [database unarchive, on page 41](#) in the *Command Reference Manual* for the syntax.

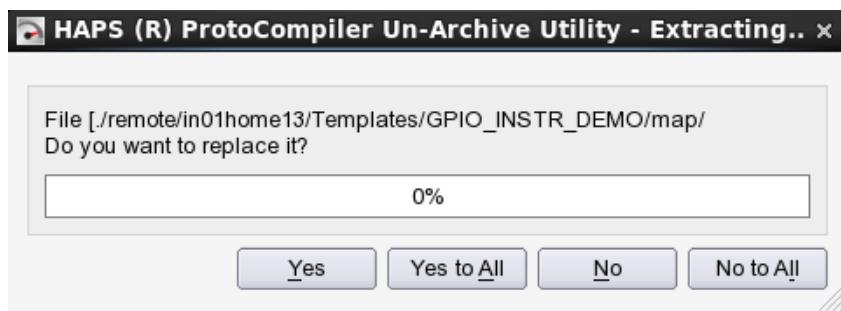
2. Specify the following information in the dialog box that opens:
  - Specify the name of the sar file containing the database files in Archive Filename.
  - Specify a location for the extracted files in Destination Directory. If you do not specify a directory, the archive file is written to the current directory.



- Click Next. This displays Screen 2.
3. Do the following in Screen 2:
- Review the summary list of files to be extracted that is displayed.
  - To load the archived database into the software tool after it has been extracted, check the Load database into ProtoCompiler check box.
  - Click Unarchive.



If the destination directory contains project files with the same name as the files you are extracting, a dialog box asks you to confirm that the existing files can be overwritten.



After extracting the files, the un-archive utility reports the status of the task, and the path location of the archive file.

- Click OK closes the dialog box.

An archive summary log file—*archive\_summary.log*—is created under the destination directory. Review this log file after you un-archive the database. Apart from the list of files, the log file provides the following details:

- Version of the ProtoCompiler with which the archive was created
- Database name
- Root directory
- Date on which the archive was created
- Number of files
- Number of not readable files
- Total Size of the database excluding the summary log file

## Creating Minimal (Lite) Archives

Whenever possible, it is recommended that you archive the entire tree, because a full archive makes it easier to duplicate and recreate results. However, if all the files cannot be archived for proprietary or other reasons, you can create a *lite* archive with the minimum of files needed.

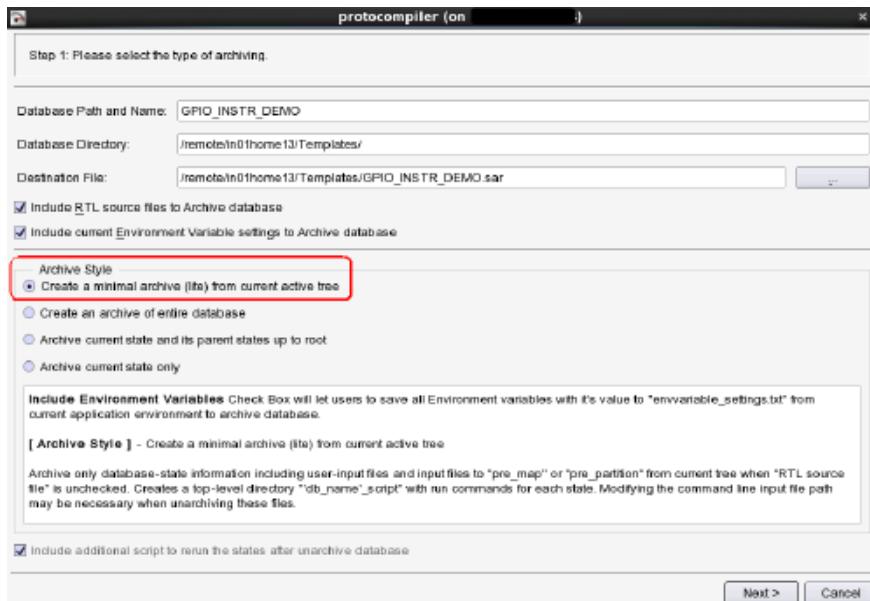
For information about creating full archives, refer to [Archiving a Database, on page 67](#). The following procedure describes how to create a minimal archive.

1. Start from a post-compile database state, like pre-partition or pre-map, and make it the active state.

The database state does not have to have passed successfully.

2. Create the minimal (lite) archive.

- Specify the database archive -mode lite command from the command line or use the GUI as described below. See [database archive, on page 32](#) in the *Command Reference Manual* for the complete command syntax.
- In the Data View, select Database -> Archive Database to open the archive utility.
- On the first page of the dialog box, set Archive Style to Create a minimal archive (lite) from current active tree.



- On the next dialog box, check the list of files and add or remove files as desired.
- Fill in the other options to create the database (see [Archiving a Database, on page 67](#) for details). To include the input RTL files, enable **Include RTL source**. Review the summary of files to be archived.

The utility archives the current state and its parents, up to root. If the current tree includes a pre-map or pre-partition state, the utility archives all the input required to rerun this state, including the compiled netlists.

You can now verify the archived files in the database, as described in the next steps.

3. Copy the db.sar archive file to a directory to extract and verify it.
4. Extract the archive files by running the database unarchive command in the Tcl window. To use run commands to generate downstream database states from a lite archive, follow these steps:
  - Unarchive the lite database, which consists of the files for the active tree but which excludes proprietary information.

- Check the script included in the archive for run commands. You might have to edit the paths to constraint files, depending on where you unarchived the lite database. You can either use the script to execute the run commands or specify them manually, as described next.
- To specify the run commands manually, start from the compiled database state and specify the next run command. Even if you archived a terminal state (e.g map, system generate), you must regenerate the intervening states from the compiled state on.

# Compiling the Design

For the context of this tool, the *compile* operation is more granular than the definition in other Synopsys verification tools: *compile* here only refers to what other tools call “front-end compile.” Subsequent design stages that make up “back-end compile” in other tools (partitioning, mapping, and place-and-route) are separate design stages in the context of this tool.

The compilation stage is where HDL is simplified and mapped to logic gates. During this stage you also validate the input files. Compilation is an iterative process.

There are two compilers available: the standard compiler or native compiler, which is built into the tool, and the unified compiler (UC), which is a shared VCS compiler front-end that is shared across many tools. It is recommended that you use UC, but older designs might use the standard compiler.

With the standard compiler flow, you typically start compiling from the root database, but you might also compile the design after instrumenting it. With the UC flow, you compile the UCDB database (see [Using Unified Compile, on page 120](#)). After compiling the design, the tool generates a compiled database that can be used for the next stage of the design.

Most of the following topics apply to compiling with the standard compiler:

- [Compiling with the run compile Command](#), on page 80
- [Running Diagnostic Compiler Mode](#), on page 85
- [Running Fast Compiler Mode \(Standard Compiler\)](#), on page 86
- [Using Different Standard Compiler Modes Together](#), on page 87
- [Running Bottom-Up Compile](#), on page 90
- [Controlling the Compiler Run with Options and Constraints](#), on page 92

See these topics for information about compiling with UC and running incremental compile:

- [Using Unified Compile](#), on page 120
- [Compiling Incrementally](#), on page 144

## Compiler Comparison

The following table compares the standard or native compiler to UC. For information about using the native compiler, see *Creating and Compiling Databases > Compiling the Design* in the *HAPS® Prototyping User Guide*.

| Native Compile (Standard Compile)                                                                                                       | Unified Compile                                                                                                                                                                                    |
|-----------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Compile refers to front-end compile only, and this stage is distinguished from subsequent design stages like pre-map and map.           | In other tools that use the same VCS compiler, <i>compile</i> includes “back-end” stages like mapping and partition. With this tool, <i>compile</i> consistently refers to front-end compile only. |
| Uses top-level module and Verilog standard specified by the <code>-top_module</code> and <code>-vlog_std</code> arguments respectively. | Uses the top-level module and Verilog standard defined in the Unified Compile database (ucdb).                                                                                                     |
| Supports VHDL and mixed language                                                                                                        |                                                                                                                                                                                                    |
| Supports UUM (Library Mapping Files) to match the VCS software when targeting HAPS systems. Sets up the software for UUM using LMF.     | Re-uses the existing VCS compilation scripts to target HAPS hardware.                                                                                                                              |
| Native front-end compiler analyzes and elaborates the RTL files.                                                                        | The VCS tool analyzes and elaborates RTL files and generates a word-level netlist.                                                                                                                 |
| Supports UPF 1.0 and 2.0 constructs defined in UPF file.<br>Supports the standard Tcl syntax.                                           | UPF is processed by VCS (unified UPF).                                                                                                                                                             |
| Supports both RTL and post-compile debug instrumentation                                                                                |                                                                                                                                                                                                    |
| Supports predefined macros and compilation directives in RTL.                                                                           | Use the VCS command line options to define all synthesis macros and compiler directives.                                                                                                           |
| Synthesis attributes can be specified as comments in the RTL as well as in a CDC file.                                                  |                                                                                                                                                                                                    |
| Supports formal verification with Formality.                                                                                            |                                                                                                                                                                                                    |
| User-Defined Primitives (UDP) are not supported                                                                                         | UDPs are supported                                                                                                                                                                                 |
| EDIF/NGC input supported                                                                                                                | EDIF/NGC input is supported as input to run compile with the ucdb option; not supported in the VCS script.                                                                                         |

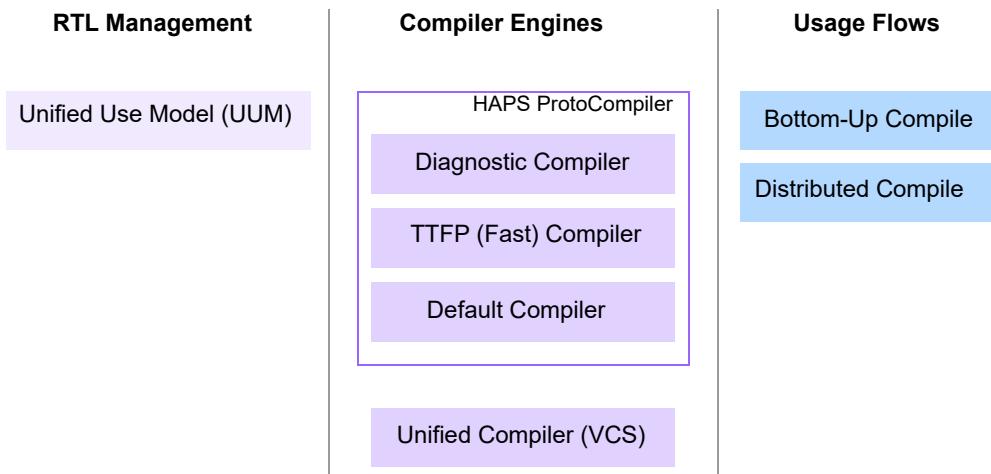
## Native Compile (Standard Compile)    Unified Compile

|                                                                 |                                                                                                                                                                                                   |
|-----------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| P1735 and Synopsys encryption (synenc) standards are supported. | Supported if encryption is handled by the VCS tool: P1735 and synenc are supported with the VCS key). For more about encryption, refer to the chapter in the VCS <i>User Guide</i> on encryption. |
| Source-level partitioning supported.                            | Source-level partitioning not supported.                                                                                                                                                          |
| Monolithic/distributed compiler mode supported.                 | Only distributed compiler mode supported.                                                                                                                                                         |

## Standard Compiler Modes and Flows

The tool has three native compiler modes that can be used in combination and also run incrementally: the default compiler, the diagnostic compiler, and the fast compiler. This is only for the standard compiler and does not apply to the UC flow ([Using Unified Compile, on page 120](#)).

Widely differing aspects are often referred to as *compiler modes* or *compiler flows*. For example, you can run compile processes in distributed mode (distributed compiler mode). The following figure and table draws a distinction between the actual compiler engines available with the native compiler (modes) and other techniques or usage:



| Name                                             | Description                                                                                                                                                                                                                                                                |
|--------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Default or native compiler                       | Compiler for better timing.<br>option set synthesis_strategy advanced<br>run compile<br><a href="#">Compiling with the run compile Command</a> , on page 80                                                                                                                |
| Diagnostic compiler                              | Compiler check, run initially to parse the RTL and CDC files and identify errors.<br>report rtl_diagnostics<br><a href="#">Running Diagnostic Compiler Mode</a> , on page 85                                                                                               |
| Fast or TTFFP (time to first prototype) compiler | Fast compiler mode with fewer optimizations, run as a first pass for RTL pipecleaning.<br>option set continue_on_error 1<br>option set synthesis_strategy fast   routability<br>run compile<br><a href="#">Running Fast Compiler Mode (Standard Compiler)</a> , on page 86 |
| Unified compiler (UC)                            | Reuses RTL database generated in the VCS environment.<br>launch uc -utf<br>run compile -ucdb<br><a href="#">Using Unified Compile</a> , on page 120                                                                                                                        |
| Unified Use Model (UUM) compiler flow            | Syntax to integrate overlapping IP modules or file names by associating HDL files, macros, defines, and include paths with a specific library.<br><a href="#">Using UUM and Group Mapping</a> , on page 109                                                                |
| Bottom-up compiler flow                          | A divide-and-conquer usage strategies, recommended for complex designs or for IPs, where modules can be compiled separately and then stitched together.<br><a href="#">Running Bottom-Up Compile</a> , on page 90                                                          |
| Distributed compiler mode                        | Processing technique to distribute compiler jobs and reduce runtime.<br><a href="#">Running Distributed Compile</a> , on page 503                                                                                                                                          |

## Compiling with the run compile Command

Compiling is an iterative process. The goal at this point is to assign the RTL logic and to rid the input RTL files of any errors, so that the operations that are later in the flow run smoothly.

The following procedure shows you the basics of running the default compiler as a separate step, but you can also run it as a sub-step of the entire synthesis process based on design goals (see [Synthesizing Based on Design Intent, on page 493](#)), or include the command in a script and run it from there.

1. Prepare the design for compilation.

- Make sure your design is FPGA-ready. See [Chapter 2, Converting ASIC Designs to FPGA](#) for a discussion about SoC-to-FPGA migration issues.
- For the standard compiler flow, make sure to specify the source files for the design. Run preliminary syntax and synthesis checks on the source files as described in [Specifying Source Files \(Standard Compiler\), on page 97](#).

Go to the design database directory that you want to compile.

Typically, you start compiling from the root database, but you might also compile the design after instrumenting it.

- For the UC flow, create the ucdb database as described in [Preparing the Input for Unified Compile, on page 125](#) and [Using Unified Compile, on page 120](#).

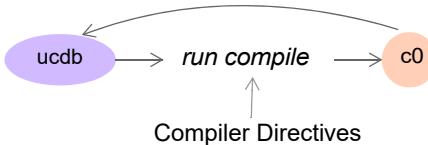
2. Specify directives or other options for the compiler.

For the standard compiler, the constraints you enter are source migration constraints and HDL constraints. For the UC flow, you cannot set HDL constraints as that is handled through the VCS front-end.

- Create a cdc file that contains compiler directives. See [Controlling the Compiler Run with Options and Constraints, on page 92](#) for a list of directives, and [Specifying Directives in a CDC File, on page 168](#) for information on how to specify them.
- Create a Tcl file for other options by typing the `edit` command in the Tcl window. For example: `edit compileOpts.tcl`.
- Use Tcl syntax and the `option set` command to set various options for the compiler. For details about setting options, see [Setting Options, on page 171](#).
- For initial synthesis runs until you achieve clean design HDL, specify the Continue on Error (CoE) option: `option set continue_on_error`. This option enables compilation to continue with the rest of the design instead of stopping when it encounters an error.

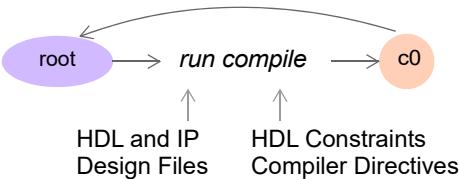
- Use relaxed constraints for initial synthesis runs.
  - To use the default standard compiler, check that `synthesis_strategy` is set to advanced.
  - Source the options file. For example: `source compileOpts.tcl`
3. To compile the design, click Compile in the GUI or specify the run compile command at the tool prompt.
- For the UC flow, start with the ucdb database generated by the launch uc command. The source files are already part of the database and do not need to be specified. For details about the UC flow, see [Using Unified Compile, on page 120](#).

```
run compile -ucdb myUCDB
```



- For the standard compiler, you can also specify Verilog include libraries and the top module at the command line with the `-include_library` and `-top_module` arguments to the run compile command, respectively.

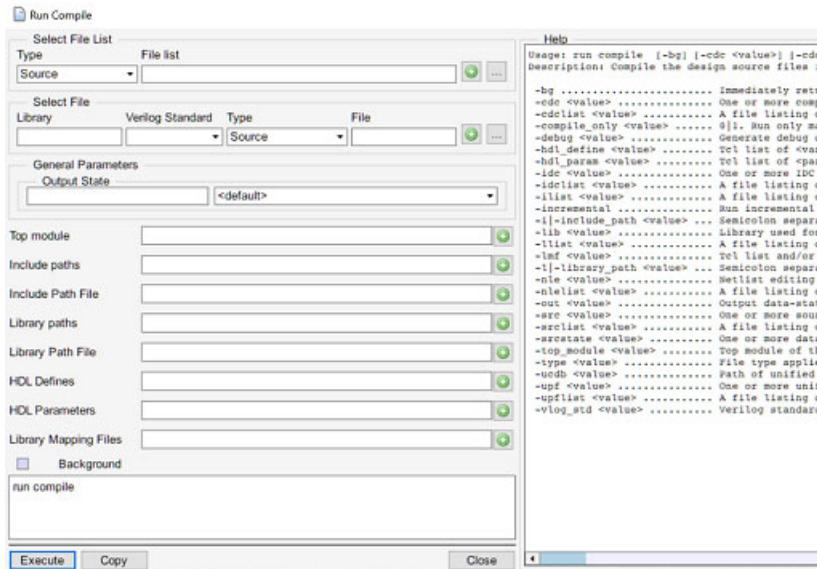
```
run compile -srclist mySrcFiles.txt -cdc const.cdc -log mydesign.log
```



The `run compile` command runs the default compiler, but you can also run other compilers, which are tuned for specific design needs. The following table lists the commands for the different compilers; see [Using Different Standard Compiler Modes Together, on page 87](#) for information on how to combine them.

| For ...                                                                                                                                                                                                  | Command                                              |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------|
| The default compiler<br>Recommended use: for best performance.                                                                                                                                           | run compile                                          |
| The diagnostic compiler<br>Recommended use: First pass to ensure that the RTL is clean, before running one of the other compilers.<br>See <a href="#">Running Diagnostic Compiler Mode , on page 85.</a> | report rtl_diagnostics                               |
| The fast compiler<br>Recommended use: for a quick prototype<br>See <a href="#">Running Fast Compiler Mode (Standard Compiler) , on page 86.</a>                                                          | option set<br>synthesis_strategy fast<br>run compile |

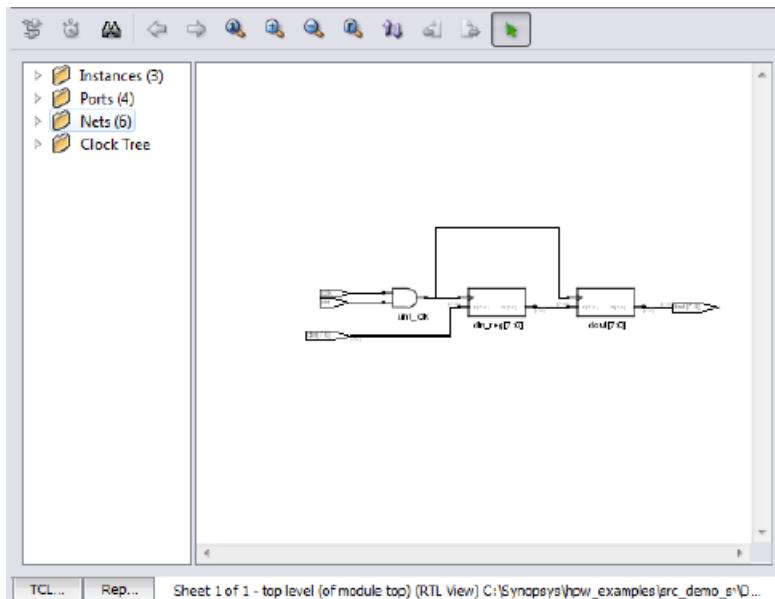
The following figure shows the corresponding GUI dialog box. See [Run Compile Dialog Box, on page 101](#) in the *Reference Manual* for descriptions of the dialog box options.



If this is the first time you are compiling the design, the software creates a new database state, called c0 by default. If you have previously compiled the design, the tool creates a new parallel database state under root, and gives it a sequential name if you do not specify one: c1. The tool also generates a schematic and a report, which you can use to analyze the results of the run.

4. Check the results and fix any errors.

- Check the schematic and the log file. See [Analyzing Results, on page 512](#) for more information about checking the results of the compilation run.
- Run the constraint checker (report constraint\_check) and clean up constraints. It is recommended that you do this to validate the constraints before running pre-map and map. For details about running the constraint checker, see [Specifying Directives in a CDC File, on page 168](#).



5. Rerun compilation as many times as needed.

You can run a combination of different compilers. See [Using Different Standard Compiler Modes Together, on page 87](#) for a recommended flow.

6. To run the default compiler incrementally on subsequent runs, specify -incr 1 with the run compile command.

The next step after compilation depends on what you want to do next:

Proceed with an initial implementation without debug [\*Running Pre-Map , on page 474\*](#)

Instrument the design for later debug [\*Instrumenting the Design for Debug , on page 641\*](#)

Partition the design [\*Setting up Files for Partitioning , on page 183\*](#)

## Running Diagnostic Compiler Mode

Run the diagnostic compiler as a first step, before running any other compiler mode. Refer to [\*Using Different Standard Compiler Modes Together, on page 87\*](#) for guidelines on using different compilers.

Use the diagnostic compiler to parse the RTL and check that it is synthesizable, that it is error-free, and that it does not contain missing modules.

1. Start with the root database and specify the report rtl\_diagnostics command. The tool runs the diagnostic compiler instead of the default compiler.

The diagnostic compiler generates a diagnostic report, but it does not generate a schematic or a c0 database state. As no database is generated, you cannot progress to the next step with this command, but you must run one of the other two compilers to generate a compiled database and proceed with the flow.

2. Analyze and fix any errors that the diagnostic compiler identified.
3. On subsequent runs, you can run the command incrementally with the -incr 1 argument to the run compile command.

Once the RTL is free of errors, you can run the default compiler or the fast compiler, according to your design needs. [\*Compiling with the run compile Command, on page 80\*](#) or [\*Running Fast Compiler Mode \(Standard Compiler\), on page 86\*](#) for details.

## Running Fast Compiler Mode (Standard Compiler)

The Time-to-first-prototype (TTFP) mode is a fast compiler mode that is designed for quick prototyping when your goal is a fast turnaround time. Use this compiler for initial runs, or when you want to generate a prototype quickly to validate basic feasibility.

1. Run the diagnostic compiler and fix any RTL errors, as described in [Running Diagnostic Compiler Mode, on page 85](#).
2. When you are satisfied with the RTL, run the fast compiler:
  - To specify fast compiler mode, set this option:

`option set synthesis_strategy fast`

If you specified the option in a Tcl options file, source the file.

The `synthesis_strategy fast` option applies to both compilation and mapping, so where you specify the option is important. If you want to run the default compiler and perform fast mapping, specify `synthesis_strategy fast` at the mapper database state. If you specify it at the root level, the setting applies to compilation as well as mapping, and the tool uses the TTFP compiler.

- Specify the `run compile` command.

The tool compiles the design using the Time-to-first-prototype (TTFP) mode, which is designed for initial prototyping and fast turnaround times. If this is the first time you are compiling the design, the software creates a new database state, called `c0` by default. If you have previously compiled the design, the tool creates a new parallel database state under `root`, and gives it a sequential name if you do not specify one: `c1`. The compiler generates a schematic which you can view to graphically analyze the design.

You can also run the fast compiler by setting your design goal in the tool window. See [Using Design Intent to Synthesize a Quick Prototype, on page 493](#) for details.

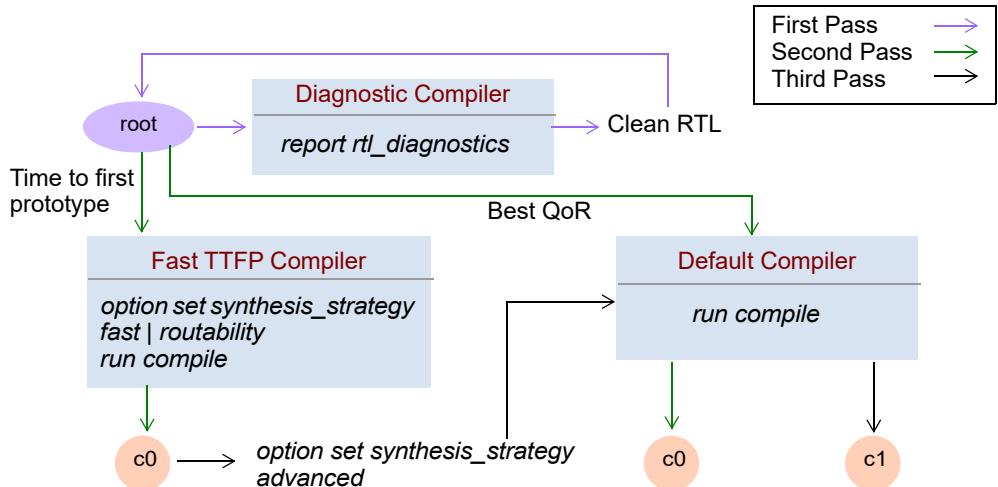
3. You can run compilation multiple times, using different compiler modes. To run a compiler incrementally on subsequent runs, specify `-incr 1` option with the `run compile` command.

The next step after compilation depends on what you want to do next:

|                                                      |                                                                  |
|------------------------------------------------------|------------------------------------------------------------------|
| Proceed with an initial implementation without debug | <a href="#">Running Pre-Map , on page 474</a>                    |
| Instrument the design for later debug                | <a href="#">Instrumenting the Design for Debug , on page 641</a> |
| Partition the design                                 | <a href="#">Setting up Files for Partitioning , on page 183</a>  |

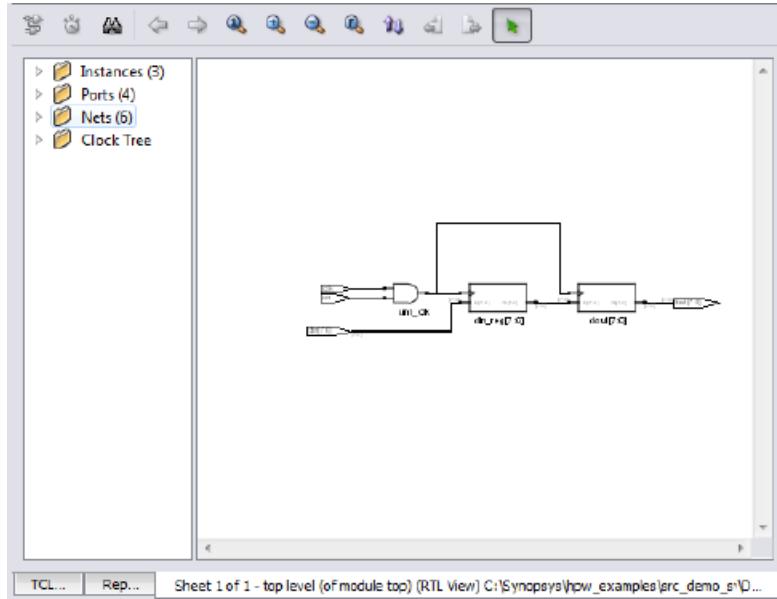
## Using Different Standard Compiler Modes Together

The prototyping tool includes three compiler modes, so that you can use the appropriate one to target your design goals. The following figure shows how you can use all three together to leverage their strengths and fit them into your design strategies.



- To pipe-clean the RTL at the beginning of prototyping, run the diagnostic compiler with the `report rtl_diagnostics` command.
  - Specify the `report rtl_diagnostics` command. The tool runs the compiler instead of the default compiler.
 Use this mode to parse the RTL and check that it is synthesizable, that it is error-free, and that it does not contain missing modules. The diagnostic compiler generates a diagnostic report, but it does not

- generate a schematic or a premap database state. If you run multiple runs, you can run the command incrementally with the -incr 1 option.
- Check the results in the log file and fix any errors.
  - Run the regular compiler with the run compile command when you are satisfied with the results of the diagnostic run.
2. For the second pass, use the fast TTFP compiler to create a quick initial prototype, as follows:
- Start with clean RTL (step 1).
  - Set option set synthesis\_strategy fast|routability and option set continue\_on\_error 1.
  - Compile using run compile. The tool uses the TTFP compiler instead of the default compiler, and does not do as many compiler optimizations. It black boxes any modules with errors and continues with the rest of the design.
- If this is the first time you are compiling the design, the software creates a new database state, called c0 by default. If you have previously compiled the design, the tool creates a new parallel database state under root, and gives it a sequential name if you do not specify one: c1.
- Fix errors and recompile with run compile -incr 1.
3. For the third pass, compile for an at-speed prototype using the default compiler:
- Start with clean RTL (step 1).
  - Make sure these options are set:  
option set synthesis\_strategy advanced  
option set continue\_on\_error 1
  - Compile using run compile. The tool uses the default compiler, which is performance-oriented. It black boxes any modules with errors and continues with the rest of the design. See [run compile, on page 132](#) in the *Command Reference* manual for the complete command syntax.
- In addition to the compiled database (c0), the tool also generates an srs file, which can be viewed in a schematic window to analyze the results of the run.
- Analyze the results. See [Analyzing Results, on page 512](#) for more information about checking the results of the compilation run.



- Fix errors and recompile with `run compile -incr 1`.
- 4. To compile for an at-speed prototype using both the TTFP and default compilers, start with clean RTL and then do the following:
  - Run the TTFP compiler as described in step 2.
  - Run the default compiler as described in step 3.
- 5. You can run compilation multiple times, using different compilers as needed. To run a compiler incrementally on subsequent runs, specify the `-incr 1` option with the `run compile` command.

The next step after compilation depends on what you want to do next:

Implement single-FPGA design  
without debug

[Running Pre-Map , on page 474](#)

Instrument the design for later  
debug

[Instrumenting the Design for Debug , on  
page 641](#)

Partition the design into multiple  
FPGAs

[Setting up Files for Partitioning , on  
page 183](#)

## Running Bottom-Up Compile

Bottom-up compilation is a useful technique for managing complex designs with divide-and-conquer strategies, to distribute and divide work between teams, to incorporate IP, or to isolate sections of designs. It also helps with interoperability, to migrate designs from other tools like VCS or Design Compiler®.

Run bottom-up compile by first compiling individual blocks and then stitching them together and running compile on the top level. You can run bottom-up compilation on a single database where you stitch together multiple compile database states, or stitch together compiled database states from different databases.

1. Compile the individual blocks, with the blocks as the top level.

The following snippet compiles two blocks, ip1 and ip2.

```
Compile ip1 as compile state c1 in database ip1
database load ip1 -autocreate
run compile -srclist f1.txt -top_module a -out c1
Compile ip2 as compile state c2 in database ip2
database load ip2 -autocreate
run compile -srclist f2.txt -top_module b -out c2
```

If the design has module instances that are nested in various modules that will be stitched together, make sure that the parent module includes definitions for all the instantiated modules it contains.

If a top-level module, TOP, contains instances of modules A and B, and B contains a nested instance of C, then TOP must include module definitions for A and B, and B must include the module definition for C. If the definition for the instantiated sub-module is not included in the parent module, there might be black box issues during place and route.

2. Compile the top level separately.

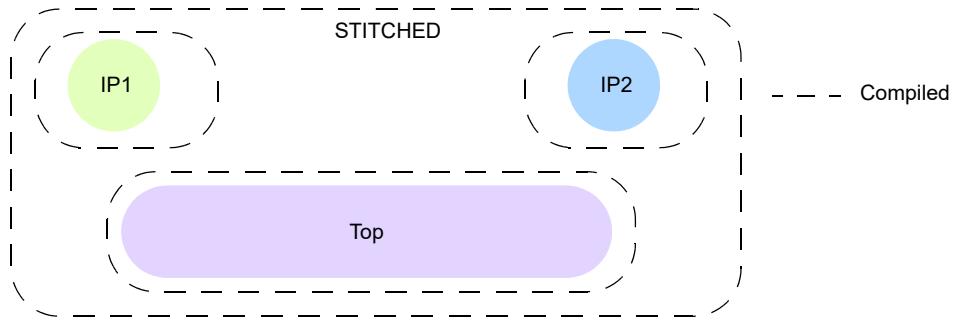
```
Compile the top level as compile state c3 in database top
database load top -autocreate
run compile -srclist add_files_top.txt -top_module top -out c3
```

The blocks can be black boxes.

3. Stitch the compiled blocks and top level together by running compile on all the elements, using the -type srcstate command argument.

This example stitches together compiled states from different databases (ip1 and ip2) into one stitched database (top\_bu\_stitched).

```
Stitch blocks and top level together in one database state
database load top_bu_stitched -autoCreate
run compile -srcstate ip1|c1 -srcstate ip2|c2 -srcstate top|c3
-top_module top -out top_bu_stitched
```



In the syntax, separate the path from the name of the database with a pipe (|). For example: home/myDesign/ip1|c0. You cannot enter the srcstate argument from the GUI.

The example creates a new database called top\_bu\_stitched, which stitches together the netlists for ip1 with the c1 compile state and ip2 with the c2 compile state. The netlist for the top\_bu\_stitched database has all the netlists stitched together, with no black boxes. The command only stitches obfuscated databases together; for other databases, include them with the -src or -srclist arguments, as described in the next step.

Note that you can only use stitching for databases from the J-2015.03 and later releases. To link in and compile netlists from previous versions of the tool or from the Synopsys FPGA synthesis tools, define them in the source file list.

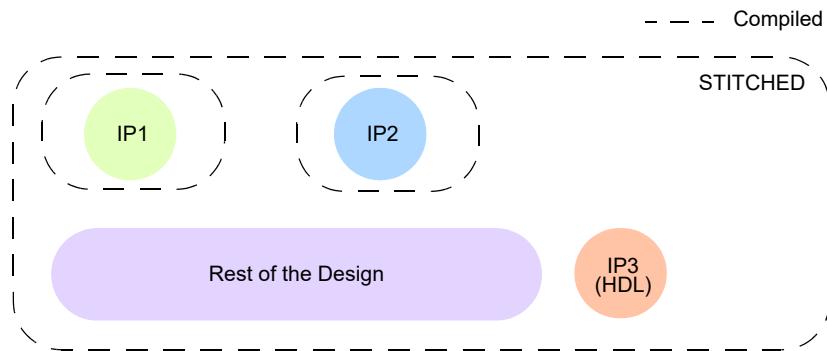
4. If you have a mix of compiled databases and HDL or non-obfuscated databases, use the -src or -srclist arguments to stitch them together, as shown below.

The figure shows a design with IP1 and IP2 as compiled databases and IP3 as an HDL database.

```
Compile ip1 as compile state c1 in database ip1
database load ip1 -autocreate
run compile -srclist f1.txt -top_module a -out c1

Compile ip2 as compile state c2 in database ip2
database load ip2 -autocreate
run compile -srclist f2.txt -top_module b -out c2

Stitch ip1, ip2, ip3, and the top level in one database state.
Source file list includes ip3.v for IP3
database load top_bu_stitched -autocreate
run compile -srcstate ip1|c1 -srcstate ip2|c2 -srclist myFiles.txt
-top_module top -out top_bu_stitched
```



## Controlling the Compiler Run with Options and Constraints

You can manipulate the results of the compiler through these controls:

- Specify arguments to the `run compile` command. For example, use the `hdl_define` argument to specify design parameters and compiler directives normally specified with `'ifdef` and `'define` statements in Verilog designs. See [run compile, on page 132](#) in the *Command Reference Manual* for the complete syntax.
- Add compiler directives to a Tcl file with a `.cdc` extension. Specify them with the `directive` command, according to the language used for the module. See [Specifying Directives in a CDC File, on page 168](#) for details.

|         |                                                                                                    |
|---------|----------------------------------------------------------------------------------------------------|
| Verilog | <code>define_directive {v:[libraryName.]entityName[(architectureName)]} {directive} {value}</code> |
|---------|----------------------------------------------------------------------------------------------------|

|      |                                                                                |
|------|--------------------------------------------------------------------------------|
| VHDL | <code>define_directive {v:[libraryName.]moduleName} {directive} {value}</code> |
|------|--------------------------------------------------------------------------------|

- Add option set command options in the Tcl window or in a Tcl file. See [Setting Options, on page 171](#) for details about working with options.

## Examples of Compiler Options and Constraints

| Option/Constraint                                                                     | Description                                                                                                                                                                                                                                                          |
|---------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Compiler Directives (cdc File)                                                        |                                                                                                                                                                                                                                                                      |
| <code>syn_black_box name</code>                                                       | Defines a module as a black box for synthesis.                                                                                                                                                                                                                       |
| <code>syn_no_prune 1 0</code>                                                         | Determines whether unused output ports on instances and black-box modules, including technology-specific primitives, are optimized away. The default is 0, where the ports can be optimized.                                                                         |
| <code>syn_preserve 1 0</code>                                                         | Prevents sequential optimizations like constant propagation, inverter push-through and FSM extraction for the specified module or entity. The default is 0.                                                                                                          |
| <code>syn_sharing off   on</code>                                                     | Specifies whether resources are shared during the compile stage.                                                                                                                                                                                                     |
| <code>syn_rename_module name</code>                                                   | Renames a submodule to avoid naming conflicts between a generated name and the original name.                                                                                                                                                                        |
| <code>syn_unique_inst_module name</code>                                              | Renames an instance of a module without affecting the other instances.                                                                                                                                                                                               |
| Option Settings (option set). Click the Options Editor icon for a comprehensive list. |                                                                                                                                                                                                                                                                      |
| <code>auto_infer_blackbox 0 1</code>                                                  | Determines whether the tool errors out when it encounters an undefined Verilog module. The default (0) causes the compiler to error out, but when set to 1, the tool creates a black box for the undefined module, issues a warning, and continues with compilation. |
| <code>automatic_compile_point 0 1</code>                                              | Determines whether the tool automatically divides up the design into compile points, without creating additional hierarchy. Compile points are RTL partitions of the design that are treated as independent blocks by the synthesis engine.                          |

| Option/Constraint                   | Description                                                                                                                                                                                                                                                                         |
|-------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| continue_on_error 1 0               | When enabled (1), directs the compiler to continue after encountering a non-syntax error and to continue compiling the next design unit without stopping. The default is 1. See <a href="#">Using Continue on Error , on page 820</a> for details.                                  |
| dc_root                             | Specifies the path to the Design Compiler installation for access to Synopsys foundation and minPower DesignWare libraries. The default is \$SYNOPSYS.                                                                                                                              |
| allow_duplicate_modules 1 0         | Allows duplicate module names to be used when set to 1. The software uses the last definition of the module and ignores any previous definitions. The default is 0.                                                                                                                 |
| dw.foundation 1 0                   | Specifies the Synopsys DesignWare foundation library as the source of the DesignWare building blocks. The default is 1. If you also specify dw_minpower, the tool overlays the DesignWare minPower library over the foundation library.                                             |
| dw_minpower 1 0                     | When set to 1, uses the DesignWare minPower library as the source for the DesignWare building blocks. The tool replaces DesignWare foundation library building blocks with the corresponding blocks from the minimum power library. The default is 0.                               |
| dw_stop_on_nolic 1 0                | Specifies whether compilation stops with an error message when the design contains a Synopsys DesignWare building block but there is no license for it. The default (1) is for the tool to stop with an error message. When set to 0, the tool black boxes the block and continues. |
| enable_prepacking 1 0               | Prepares the netlist for LUT combining during place-and-route, so that the Xilinx tool can pack into the LUT6_2 depending on available resources                                                                                                                                    |
| synthesis_strategy fast routability | Runs the compiler in a fast mode that is optimized for prototyping. This mode does not run compiler optimizations. See the <i>Reference Manual</i> for a description.                                                                                                               |
| hdl_param -set option               | Sets HDL parameter overrides for VHDL designs                                                                                                                                                                                                                                       |

| Option/Constraint                                    | Description                                                                                                                                                                                                                                                                                                                                                   |
|------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>include_path path</code>                       | Defines the search path used by the ‘ <code>include</code> ’ commands in Verilog design files. <i>path</i> is a semicolon-delimited list of directories where the included design files can be found. The include paths are relative. The software searches for include files in the following order, and stops at the first occurrence of the included file: |
|                                                      | <ul style="list-style-type: none"> <li>• The source file directory</li> <li>• Files in <i>path</i>, in the specified order</li> <li>• The project directory</li> </ul>                                                                                                                                                                                        |
| <code>libext .libextName1<br/>.libextName2 ..</code> | Specifies additional library extensions for Verilog library files in addition to <code>.v</code> and <code>.sv</code> ; for example: <code>.av</code> , <code>.bv</code> , <code>.cv</code> , <code>.xxx</code> , <code>.va</code> , <code>.vas</code> . The tool searches the directory paths you specified for Verilog library files with these extensions. |
| <code>library_path path</code>                       | Specifies the paths to the directories which contain Verilog library files. The tool uses <i>path</i> to find all included library files and determine the top-level module.                                                                                                                                                                                  |
| <code>looplimit limit</code>                         | Lets you override the default compiler loop limit value of 2000.                                                                                                                                                                                                                                                                                              |
| <code>max_parallel_jobs<br/>num_of_jobs</code>       | Determines the maximum number of processing jobs to run in parallel. Each license allows up to four parallel jobs. Use this in conjunction with <code>automatic_compile_point</code> to reduce runtime.                                                                                                                                                       |
| <code>top_module name</code>                         | Specifies the top-level entity, when the top-level entity does not use the default work library to compile the VHDL files. <i>name</i> is the VHDL library, followed by a dot (.) and the name of the top-level entity.                                                                                                                                       |
| <code>vhdl2008 1 0</code>                            | Overrides the default VHDL standard (VHDL), and sets the standard to VHDL 2008.                                                                                                                                                                                                                                                                               |
| <code>vlog_std v95   sysv</code>                     | Sets the Verilog standard to Verilog 95 or SystemVerilog, instead of the default, v2001.                                                                                                                                                                                                                                                                      |

# Adding Design Files

When you specify design files, they apply to the current database state and all states below them. This table summarizes the different kinds of files:

|                                          |                                                                                                                                                                                                                                                                                                                                                                                                                          |
|------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Source Files<br>(Standard Compiler only) | You must specify them at the first implementation phase, after you have created the database. Generally, you specify the files as an argument (-src, -srclist) to the appropriate run implementation command for an initial design phase (run_compile or run_pre_instrument). See <a href="#">Specifying Source Files (Standard Compiler)</a> , on page 97 and <a href="#">Using UUM and Group Mapping</a> , on page 109 |
| Tcl Option Files                         | You can specify option files for any database state, instead of specifying individual options on the command line. The options specified apply to the current database state and the states below it. Some options only apply to a specific database state, and can only be specified for that state. You must source the options file for the options to apply. See <a href="#">Setting Options</a> , on page 171.      |
| Constraint Files                         | There are different constraint files. Some, like the cdc file, can only be specified for specific database states to which they apply. Generally, you specify these files as an argument to the implementation command for that database state. See <a href="#">Specifying Constraints</a> , on page 167                                                                                                                 |

For details, see these topics:

- [Specifying Source Files \(Standard Compiler\)](#), on page 97
- [Using UUM and Group Mapping](#), on page 109
- [Setting Options](#), on page 171
- [Specifying Constraints](#), on page 167
- [Converting Legacy Projects to Databases](#), on page 65

# Specifying Source Files (Standard Compiler)

Design source files can be the same as or derived from ASIC source files. For details about the differences in FPGA design that the design files must take into account, see [Getting ASIC Designs Ready for FPGA, on page 34](#) and [Handling I/O Pads, on page 37](#).

Once the source files are ready, the first step is to add them to the design. For the standard compiler flow, there are different ways to do this:

As part of a run command for an initial design phase (compile, or pre-instrument) with the -src or -srclist arguments.

[Specifying Source Files Using a Command Argument, on page 97](#)

Using a text file that lists all the source files.

[Creating a Text File List of Source Files, on page 101](#)

Through the GUI at an initial design phase (Compile, Instrumentation Preparation).

[Specifying Source Files from the GUI, on page 103](#)

With group mapping files, when you want to use the same libraries for other tasks, like VCS simulation.

[Using UUM and Group Mapping, on page 109](#)

In addition to the methods for adding source files, this section also discusses these related topics:

- [Checking for Unused Files, on page 105](#)
- [Using Variables in File Paths, on page 105](#)
- [Specifying Verilog Standards, on page 107](#)

## Specifying Source Files Using a Command Argument

The following procedure describes how to add source files to your database using command arguments. This procedure only applies to the standard compiler flow.

1. To specify the RTL files at the command line, use the -src argument with one of the initial commands (run compile, report rtl\_diagnostics, run pre\_instrument).

The following examples show several ways to specify multiple RTL files:

```
run compile -src {top.v other.v another.v}
run compile -src "$RTL_PATH/core.v $RTL_PATH/another_core.v"
run compile -src top.v -src other.v -src another.v
```

Alternatively create a file with a list of input files, as described in step 2. See [Using Variables in File Paths, on page 105](#) for details about using variables.

2. To specify the RTL files using a text file, follow these steps:

- Create a text file with one source file per line. You can have source files in a mixture of RTL formats. For details about creating this file list, see [Creating a Text File List of Source Files, on page 101](#).
- When the text file is complete, specify it with the -srclist option when you compile the design with one of the initial commands. For example:

```
run compile -srclist myFiles.txt
```

3. Add other libraries as needed, so that the files are compiled with the design.

- Set the libext option to define the file extensions for Verilog files you want to add:  

```
option set libext {.libextension1 .libextension2 ...}
```
- Specify libraries through arguments to one of the initial design commands: run compile and run pre\_instrument. For the complete syntax for these commands, see [Shell Command Reference, on page 17](#) in the *Command Reference Manual*.

The following table summarizes the run compile, report rtl\_diagnostics, and run pre\_instrument command arguments to use:

| To Add ...                                    | Command Argument                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-----------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Verilog files from specified libraries        | <p>Specify the path to the directory that contains the Verilog files. Arguments:</p> <ul style="list-style-type: none"> <li>• <code>-l path</code> or <code>-library_path path</code> for a library. Make sure that the names of the files in the path match module names.</li> <li>• <code>-list file</code> for a file that lists one or more libraries.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                       |
| Verilog include files                         | <p>Specify a list of search paths to directories that contain the include files, separating entries with semi-colons. Arguments:</p> <ul style="list-style-type: none"> <li>• <code>-i {pathList}</code> or <code>-include_path {pathList}</code> for a list of search paths to the directories that contain the include files. For example:<br/> <code>run compile -src filename -i {./path1;./path2;./path3}</code><br/>           You can also define a variable for the include path and then specify the variable:<br/> <code>set INCLUDE_PATH {./path1;./path2;./path3}</code><br/> <code>run compile -src filename -include_path \$INCLUDE_PATH</code></li> <li>• <code>-list file</code> for a file that lists one or more include paths</li> </ul> |
| Verilog user-defined primitives (UDP)         | See <a href="#">Including Verilog UDP Files , on page 100</a> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| Verilog Library mapping files (UUM libraries) | <code>-lmf</code><br>The tool adds files with the specified extensions from these directories into the list of source files. See <a href="#">Using UUM and Group Mapping , on page 109</a> for details about creating a library mapping file.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| DesignWare components                         | Do not add the DW_* files, but include DC root in your search path. If you include the DW_* files, you can get messages about errors in encrypted blocks when the design is compiled.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Search order and library binding information  | <p><code>-hdl_define</code><br/>           Lets you specify various directives.<br/>           Use <code>__SYN_COMPATIBLE_INCLUDEPATH__</code> to specify that the tool use the same search path order as the VCS tool (current directory and then include paths).<br/>           For information about using <code>__ISOLATE_LIB__</code> and <code>__USELIBGROUPORDER__</code> for lmf files, see <a href="#">Setting Library Options with -hdl_define , on page 114</a>.</p>                                                                                                                                                                                                                                                                             |

When the design is compiled, the software searches the listed directories and adds Verilog source files with the specified extensions as source files to the design.

4. To identify files that are not used for synthesis, use the `dh_module_sources` command, as described in [Checking for Unused Files, on page 105](#).

You can then remove these files from the source file list.

5. To override the default Verilog standard (v2k) or a default set in an options file, use the `-vlog_std` option to the `run compile`, `report rtl_diagnostics`, or `run pre_instrument` commands.

See [Specifying Verilog Standards, on page 107](#) for details about setting the standard and precedence order and [Working with Tcl Options Files, on page 171](#) for information about using options files.

6. Check the design files for errors.

- Use the `report syntax_check` command to run the syntax checker on the HDL files and identify syntax errors.
- Fix syntax and synthesis errors before proceeding.

Typically, the next step is to compile the design, but you could choose to instrument the design instead:

| To ...                | See ...                                                          |
|-----------------------|------------------------------------------------------------------|
| Compile the design    | <a href="#">Compiling the Design, on page 77</a>                 |
| Instrument the design | <a href="#">Instrumenting the Design for Debug , on page 641</a> |

## Including Verilog UDP Files

Verilog allows you to create user-defined primitives (UDP) to define more complex primitives than the standard Verilog ones. To include these files, do the following:

1. Create a UDP file.

```
//Example of structure of UDP body (udp_prim.v)
```

```
primitive udp_prim
a, //Port a
b, //Port b
c, //Port c
);
output a;
input b,c;

//UDP function code here
//A = B|C;

table
//B C :A
? 1 :1;
1 ? :1;
0 0 : 0
endtable

endprimitive
```

2. Instantiate the UDP in the top module:

```
module top (output logic a, input b, c);
udp_prim udp (a,b,c);
endmodule
```

3. Use VCS to parse the UDPs and generate a database.

```
vlogan -full64 -work work ./top.v/udp_prim.v -sverilog -full64
vcs -full64 work.top -top work.top -hw_top=top -full64
```

4. Read in the database generated by VCS.

Read in the database using the launch uc and run compile command sequence, as in the unified compile flow.

```
database load pcucdb -autocreate
launch uc -utf run.utf -ucdb ucdb
run compile -ucdb ucdb
```

## Creating a Text File List of Source Files

It is best to create a text file that contains a list of the source files so that you can just specify the list at the command line, rather than specifying the source files. This procedure only applies to the standard compiler flow.

1. Create a text file with one source file per line. You can have source files in these formats, or in a mixture of these formats:

|                           |                                            |
|---------------------------|--------------------------------------------|
| Verilog                   | Verilog, SystemVerilog, Structural Verilog |
| VHDL                      | VHDL, VHDL 2008                            |
| EDIF netlists             | Plain text, ngc, edn                       |
| GTECH technology netlists |                                            |

2. Use the following syntax to specify each entry.

```
[-type verilog | structver | vhdl | xilinx | edif] [-lib library]
[-vlog_std v2001 | sysv | v95] [vhdl2008 1 | 0]
fileName
```

- The only required syntax is the file name and the associated path to it; everything else is optional. See [Using Variables in File Paths, on page 105](#) for information about using variables in file paths.
- You do not have to specify -type if the file has a standard extension, because the tool automatically makes the associations. These are the standard associations:

|       |                                      |
|-------|--------------------------------------|
| .v    | -type verilog (Verilog)              |
| .vhdl | -type vhdl (VHDL)                    |
| .vm   | -type structver (Structural Verilog) |
| .edf  | -type edif (EDIF)                    |
| .ngc  | -type xilinx (Xilinx ngc)            |

- The default Verilog version is Verilog 95, but you can specify another version with the -vlog\_std argument. For example: -vlog\_std v2001 test.v. See [Specifying Verilog Standards, on page 107](#) for details about setting the standard and precedence order.

- The default library is work, but you can specify other libraries to be compiled with the -lib argument. You cannot specify the -lib argument for -type xilinx and -type edif files. Include the appropriate VHDL libraries, like third-party VHDL package libraries for example. Make sure that the package library is specified before the top-level design in the list of files:

```
src/package.vhd
src/top.vhd
```

- For GTECH generic technology components, either add the technology-independent Verilog library supplied with the software (*install\_dir/lib/generic\_technology/gtech.v*), or add your own generic component library. Do not use both together as there may be conflicts.

### Example: Source File Entries

This is an example of source file entries in a file list:

```
src/proto_haps_top.v
 -vlog_std v2001 src/haps_says_hello.v
src/gpiolink_generic.v
 -type vhdl -lib myLib ./ram_impl.vhd
src/h7_gpio_link.vhd
src/hydralink_txphy.vhd
src/hydralink_rxphy.vhd
```

## Specifying Source Files from the GUI

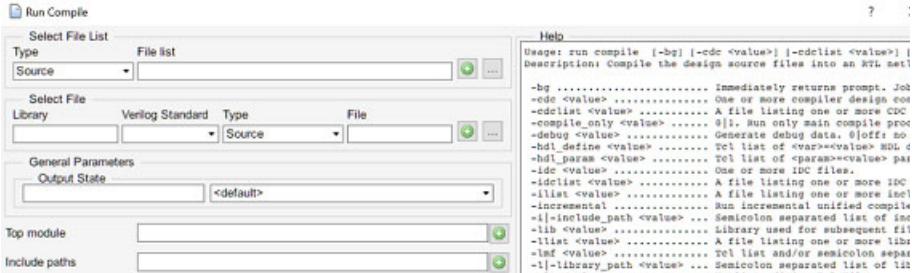
You can also specify source files from the GUI instead of the command line, but you cannot specify group mapping files from the GUI. This procedure only applies to the standard compiler flow.

1. Open the file list editor.

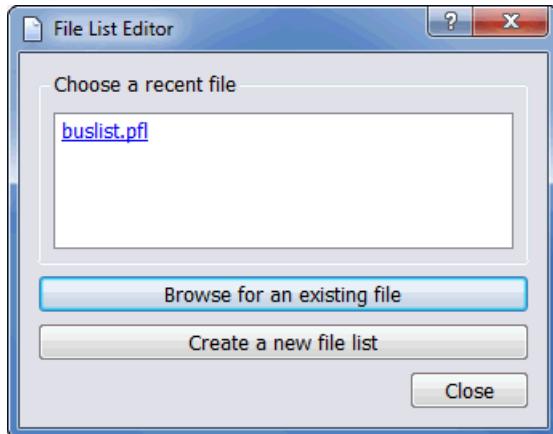
You can do this in different ways, depending on the design stage:

- Click the File List Editor icon (  ) in the main GUI window (  ). This option is available when you first start a design and are ready to compile or pre-instrument it.
- Click the File List Editor icon in the dialog box for the run compile or run pre\_instrument command you are using. The following figure shows the

relevant parts of the Run Compile dialog box, including the icon in the upper right:



When you click the icon, another dialog box opens:



2. To create a new source file list, click Create a new file list and enter the files you need, following the syntax guidelines described in [Specifying Source Files Using a Command Argument, on page 97](#).
- See [Using Variables in File Paths, on page 105](#) for information about using variables in file paths.
3. To open an existing file, either select one from the list of recent files, or click Browse for an existing file to locate one that was created previously.
  4. To identify files that are not used for synthesis, use the `dh_module_sources` command, as described in [Checking for Unused Files, on page 105](#).

## Checking for Unused Files

You can identify files that are not used for synthesis and which need not be included in the source file list, by following the steps described below. This procedure applies to the standard compiler flow.

1. Compile the design.
2. From the compiled database, use the `dm_root` command to identify the name of the top module  
For example, it could return `work.test.verilog`.
3. Use the `dh_module_sources` command with the top-level module name to generate a list of files required for synthesis:

```
dh_module_sources work.test
```

This lists the files used to build the specified top module:

```
/11_Case_dh_module/test/b.v
/11_Case_dh_module/test/a.v
/11_Case_dh_module/test/test1.v
/11_Case_dh_module/test/test.v
```

So the only files required to synthesize the top module are `test.v`, `test1.v`, `a.v`, and `b.v`. Any other files are not required and need not be added to the source files for synthesis.

You can also get a list of libraries and modules in the design after compiling it. Create a script that leverages the CG364 messages, which list the modules and libraries synthesized, to list the libraries and modules and sort them.

## Using Variables in File Paths

If you use a variable as part of a source file path, make sure to pass the variable to every new invocation of the tool executable. The variables do not persist from one session to another. However, if you use the `source` command to source a Tcl run script instead of running the `launch` command, you do not need to invoke a new executable session.

1. Set variables in one of these ways:

- As an environment variable

Specify the path as an environment variable before launching the tool:

```
set env(DESIGN_DIR) "/a/myDir"
launch protocompiler
```

- As a Tcl variable

For paths specified inside a Tcl run script, define a Tcl variable in the script before referencing it. This example defines the variable in terms of a base working directory to make the script portable:

```
set base_dir [pwd]
set DESIGN_DIR $base_dir

set OPTION_FILE $DESIGN_DIR/project_name_option.tcl
set IDC_FILE $DESIGN_DIR/project_name_debug.idc
set SRC_FILE_LIST $DESIGN_DIR//project_name_src_filelist.pfl
set XTOR_FILE_LIST $DESIGN_DIR/XTOR_src_filelist.pfl
set AXI_FILE_LIST $DESIGN_DIR/AXI_src_filelist.pfl set CDC_FILE_LIST
 $DESIGN_DIR/project_name_cdc_filelist.pfl
set FDC_FILE_LIST $DESIGN_DIR/project_name_fdc_filelist.pfl
set LIBRARY_LIST $DESIGN_DIR/project_name_library_list.pfl
set INCLUDE_LIST $DESIGN_DIR/project_name_include_list.pfl
set HAPS70_S72 $DESIGN_DIR/haps70s72_ddr3_xtor2.tss
set DATABASE_NAME project_name_270315
set RUNTIME_DIR project_name_RUNTIME_270315
set TOP_MODULE top_module_name
```

2. If you set an environment variable, you can reference that variable from inside the script:

```
source $env(DESIGN_DIR)/post_partition_options_D.tcl
```

3. Once the variable has been defined, specify source files by referencing the \$DESIGN\_DIR variable.

For example:

```
-type vhdl -lib work $DESIGN_DIR/../myFile.vhd
```

## Specifying Verilog Standards

The default Verilog standard is v2001 or v2k. This procedure only applies to the standard compiler flow.

1. To override the default, use one of the ways listed in the table below.

The table also shows the precedence, with the highest precedence first. Examples of each method follow the table.

The default Verilog standard is v2001 or v2k.

### Use `-vlog_std` in... Details...

|                                          |                                                                                                                                                                                                               |
|------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Source file list                         | Use <code>-vlog_std</code> in the source file syntax to specify the standard. See <a href="#">Source File List Example , on page 107</a> .                                                                    |
| Command line argument                    | Use the <code>-vlog_std</code> argument to the run compile, run pre-instrument, or report rtl_diagnostics commands to specify the standard. See <a href="#">Command Line Argument Example , on page 108</a> . |
| Tcl option file                          | Set a global option with option set <code>vlog_std</code> in a Tcl options file. See <a href="#">Global Default (Options File) Example , on page 108</a> .                                                    |
| <code>.synopsys_pc.setup</code> Tcl file | Set a global option with option set <code>vlog_std</code> in a Tcl startup file. See <a href="#">Global Default (Setup File) Example , on page 108</a> .                                                      |

2. To view the default Verilog standard, type option `get_default optionName`.

The command reports the value of the specified option after reading all the `.synopsys_pc.setup` files.

### Source File List Example

This setting overrides all other Verilog standard settings. In the following example, the Verilog standard for abc.sv is set to System Verilog within a file list like myFilelist.pfl.

```
-vlog_std sysv abc.sv
```

The file list can then be specified with the `-srclist` command line argument for the initial commands: run compile, run pre\_instrument, report rtl\_diagnostics.

## Command Line Argument Example

When specified with the run compile, report rtl\_diagnostics, or run pre\_instrument commands, -vlog\_std is order-dependent. In the following example, the first -vlog\_std sv argument applies to the top.sv file and sets the standard for it to System Verilog. The next -vlog\_std v95 argument applies to all the source files in the test\_src\_filelist.pfl file unless one of those listed files has its own specified standard, because the source file list specification takes precedence. The last -vlog\_std argument applies to the modQ.sv file and specifies a System Verilog standard for it.

```
run compile -top_module top -vlog_std sysv -src top.sv -vlog_std v95
-srclist ./test_src_filelist.pfl -vlog_std sysv -src modQ.sv
```

Source files specified with -library path use the global default, not the standard set by the -vlog\_std argument to the run command.

Include files use the same standard as the file that includes them, in accordance with the *Language Reference Manual* (LRM).

## Global Default (Options File) Example

When set as an option, the specified Verilog standard replaces the default v2001 standard, unless another specification takes precedence.

```
option set -vlog_std v95
```

See [Working with Tcl Options Files](#), on page 171 for more information about setting global options in Tcl files.

## Global Default (Setup File) Example

When set as an option in a Tcl .synopsys\_pc.setup file, the specified Verilog standard replaces the default v2001 standard, unless another specification takes precedence. The standard is set using an option set command in the setup Tcl file.

```
option set -vlog_std v95
```

The tool automatically reads options set in this file at startup. The location of the startup file also determines its scope, because the tool reads the files in the order specified in the table below. After reading the file, the tool uses the specified option setting as the default.

| Setup File Location                       | Directory Description                                    |
|-------------------------------------------|----------------------------------------------------------|
| <code>installDir/synopsys_pc.setup</code> | Installation. Sets the option or default in the install. |
| <code>~/synopsys_pc.setup</code>          | Sets the option or default for your personal setup.      |
| <code>./synopsys_pc.setup</code>          | Sets the option or default for the working directory.    |

To report the current default setting use `option get_default vlog_std` command.

See [Working with Tcl Options Files, on page 171](#) for more information about setting global options in Tcl startup files.

## Using UUM and Group Mapping

Group mapping allows you to use the same design files across different tools through a library mapping mechanism called the Unified Use Model (UUM). Using UUM helps interoperability between tools and optimal design management. The goal is to use the same files for both VCS designs and prototyping. UUM also serves to integrate multiple IPs from different vendors, where there might be overlapping IP module or include file names.

See the following for details:

- [About Group Mapping and the lmf File, on page 109](#)
- [Creating an lmf File, on page 110](#)
- [Group Mapping File Example, on page 113](#)
- [Default Library Mapping Example, on page 114](#)
- [Setting Library Options with -hdl\\_define, on page 114](#)

### About Group Mapping and the lmf File

Without UUM and an lmf group mapping file, the defines or macros specified from the command line are seen across the entire design regardless of the library where it was compiled. Similarly, defines or macros specified in an RTL file are accessible by all RTL files compiled after the RTL file with the defined macro, without regard to the library where it was compiled or the value of the `multi_file_compilation_unit` option.

The lmf group mapping file is a Tcl file that is used with Verilog designs to specify include paths, library directories, compiler directives and Verilog macros that are unique to a given library. The file provides guidance to resolve conflicts when there are overlapping include paths, overlapping modules in multiple libraries without a Verilog configuration file, or when there are macros in the global scope that need to be changed to the library scope. The library mapping file eliminates the need to pre-compile and merge files under a wrapper to specify unique settings, and also lets you restrict the scope of a macro to a library. It is similar to the VCS `synopsys_sim.setup` file, which is used to determine module binding when there are conflicts.

## Creating an lmf File

The following steps guide you through the creation of group mapping files, which you then specify from the command line. You cannot specify group mapping files from the GUI.

1. Create one or more group mapping files (`lmf`).

The following guidelines apply:

- You can create multiple group mapping files for a design.
- Each mapping file can provide mappings for any number of libraries.
- If a library has multiple mappings, only the last mapping is honored. The `lmf` mapping files are added in the order specified in the `-lmf` argument to the run compile or run `pre_instrument` commands.
- Ideally the `lmf` file should cover all the libraries in the design. If only a small subset of the libraries have special settings, you can use `_DefaultLibraryMapping_` to set default settings for the libraries with common settings (see the next step for details). If there are missing libraries, the tool searches the other source files outside the mapping file using this order: include path, define, library.
- Settings defined in the library take precedence. For unspecified libraries, the tool uses the regular settings for the design. For example, if you do not explicitly specify the work library in the mapping file, the tool uses setting information from the project for files located in this directory.

2. In the `lmf` file, specify the setup to be used for each logical library.

See [Group Mapping File Example, on page 113](#) for an example of this file.

- Inside the lmf file, use set\_option statements with the arguments shown in the following table to define the files and libraries:

|               |                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -include_path | Specifies different include_path options for each library. Paths are separated by semi-colons.<br>set_option -include_path {path1;path2;path3}<br>Relative paths are resolved relative to the lmf file.                                                                                                                                                                                                                              |
| -library_path | Specifies paths to Verilog library directories. The tool searches the directories for modules that are instantiated but not defined in the source code. Multiple paths are separated by semi-colons.<br>set_option -library_path {libpath1;libpath2;libpath3}<br>Relative paths are resolved relative to the lmf file.                                                                                                               |
| -hdl_define   | Defines a text macro in your source code as a value or character string. You can test for this definition in your Verilog source code using the 'ifdef compiler directive.<br>set_option -hdl_define -set {DEF1=32'hFF DEF2=1}<br>You can list multiple sets of macro=value pairs with the -hdl_define argument, separating them with spaces.                                                                                        |
| -lib_ext      | Specifies a list of file extensions for the tool to search for in a library directory. The tool searches the directories specified with library_path and include_path for files with the specified extensions.<br>set_option -libext {*.v *.sv *.v1}<br>The example specifies that the tool only search for .v or .sv and .v1 extensions in a library. The order of the file name extensions does not imply an order for the search. |

- Use \_\_DefaultLibraryMapping\_\_ to specify default settings for multiple libraries. Use this option when you have multiple libraries, but only a small proportion have individual sets of options. Specify specific settings for the individual libraries that have different settings, and use \_\_DefaultLibraryMapping\_\_ to define the common settings for the rest. See [Default Library Mapping Example, on page 114](#).
3. Specify library search order and binding options as run compile -hdl\_define options, as needed to resolve ambiguities.
- For details, see [Setting Library Options with -hdl\\_define, on page 114](#).
4. Use the single-line comment style to include comments.

```
//This is a Verilog single-line comment
```

5. Once you have created the file, add the lmf file with the -lmf argument to the run compile or run pre\_instrument commands, so that it is compiled.

For example: run compile -lmf {libmap1; libmap2}

The tool adds files with the specified extensions from the directories you specified in the lmf file, in the order specified. If there are conflicts, the tool honors the last one specified.

When the design is compiled, each IP is compiled in a separate directory, along with its library, include path and define information, so that different name spaces are maintained. The compiler reads in the lmf file, source files, and global options, and resolves them by first honoring the lmf options on a per-library basis, and then filling in any missing information in the lmf from the global options.

6. Check the results by specifying the view report command from a post-compile database state.

- This command opens the log file, where you can check the library setup information that was added. Notes like the following list the library mapping file used; check that the correct lmf file was used.

```
@N:: __SYN_GROUP_MAP__ set: reading /sample_vcs2pc/vloglib.map
@N::Reading group mapping file: /sample_vcs2pc/vloglib.map
```

- The Before parsing section of the log file contains details of the macros, include paths, library search paths, and library extensions for every library as specified in the lmf file. The After parsing has the same information as well as information about any RTL defines found. Check that the details are as expected.
- Check for errors:

| Error            | Cause                            |
|------------------|----------------------------------|
| Cannot open file | Check for missing include files. |

| Error                                | Cause                                                                                                                                                                                       |
|--------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CG389: Reference to undefined module | Check for missing library paths or libexts that result in undefined modules. Check for conflicting modules across libraries and resolve them using a config file or the libgrouporder macro |
| CG141: Creating black box for module | Check the same causes as for CG389.                                                                                                                                                         |
| Other errors                         | Check for missing or ignored defines or macros from the lmf. Check for syntax errors.                                                                                                       |

## Group Mapping File Example

Relative paths in include and library paths are resolved from the path to the lmf file. This means that in the following example, if the lmf file is in dir1/dir2, then the path to the include file defined on line 3 is dir1/dir2./ip1\_includes.

```
#Options for ip1_lib
library ip1_lib;
set_option -include_path {./ip1_includes}
set_option -library_path {./ip1_libdir}
set_option -libext .v1
set_option -hdl_define -set {DEF1=32'd0 DEF2=32'd2}
set_option -hdl_define -set {DEF11 DEF12}
endlibrary

#Options for ip2_lib
library ip2_lib;
set_option -include_path {./ip2_includes}
set_option -library_path {./ip2_libdir}
set_option -hdl_define -set {DEF1=32'hFFFFFFFFFF DEF2=32'hBAAD0000}
set_option -hdl_define -set {DEF12}
endlibrary

#Options for ip3_lib
library ip3_lib;
set_option -include_path {./ip3_includes}
set_option -library_path {./ip3_libdir}
set_option -hdl_define -set {DEF1=32'd256 DEF2=32'd257}
set_option -hdl_define -set {DEF11}
set_option -libext .v3
endlibrary
```

## Default Library Mapping Example

If the design has multiple libraries but only a small set have specific settings, (`library_path/include_path/hdl_define`) define the common settings for the remaining libraries with `_DefaultLibraryMapping_`. In this example, `lib1` and `lib2` each have distinct sets of options, but the remaining 98 libraries share the same set of options.

This is an example of the library mapping file (`lmf`) for this design scenario:

```
library lib1;
set_option -library_path ...
set_option -include_path ...
...
Endlibrary

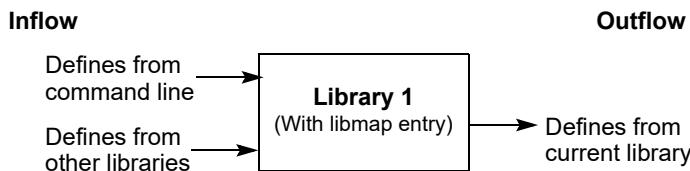
library lib2;
set_option -library_path ...
set_option -include_path ...
...
endlibrary

library _DefaultLibraryMapping_; #Applies from lib3 to lib100
set_option -library_path ...
set_option -include_path ...
...
endlibrary
```

## Setting Library Options with `-hdl_define`

The scope of macros defined with `-hdl_define` is confined to the specified library only; it is not global. Say that `temp` is specified in both `lib1` and `lib2` but is defined with `-hdl_define` in `lib1` only. Then if the HDL contains `ifdef temp ... else` code, the `ifdef` code applies to `lib1` and the `else` code applies to `lib2`.

Use `-hdl_define` macros to affect the default inflow and outflow illustrated in the figure below. If an `-hdl_define` macro is defined for the library in the `lmf` file, it overrides the default settings shown below.



- Inflow is the ability of the macro that is defined in the RTL or the global scope to be available in a library scope, provided there is no `-hdl_define` specified for that library in the lmf mapping file.
- Outflow is the ability of the macro to be available in another library scope, as long there is no `-hdl_define` for that library in the lmf file.

## Specifying `-hdl_define`

Use the `-hdl_define` argument at the run command line to specify library setup options. This argument applies to the standard compiler flow.

1. Specify `-hdl_define` as an option to the run `compile` or `run pre_instrument` commands when you add the source files.

If you specify `-hdl_define` inside the lmf file instead of at the run command line, it only applies to that library.

2. Specify the search order or library directive as the value of the `-hdl_define` argument.

- To ensure compatibility between different Synopsys tools such as Design Compiler (DC), use `SYN_COMPATIBLE`. DC ignores dynamic initialization assignments, unlike the prototyping tool. Use this macro to make DC compatible with the prototyping tool.
- To specify the same search path order as the VCS tool, use `_SYN_COMPATIBLE_INCLUDEPATH_`. Use this macro to ensure that the search order matches the search order for the simulation tool.
- To specify that the search path order be the file order listed in the lmf file, use `_USELIBGROUPORDER_`. Use this macro to define search order for conflicts when there are multiple module definitions.
- To control access to a library, use `_ISOLATE_LIB_` or `ISOLATE_LIB`. Use it to restrict access to a library or to create dummy defines.

See [-hdl\\_define Macros, on page 116](#) for details.

For example, in the following example `_USELIBGROUPORDER_` specifies that the tool use the file order listed in the lmf file:

```
run compile -srclist rtl_sources.pfl -hdl_define _USELIBGROUPORDER_
-top_module top -lmf myDesign.lmf
```

By default, in a design without group mapping, defines and macros specified at the command line are available across the entire design, regardless of the library where they were compiled. Also, defines and macros specified in HDL files are available to all HDL files compiled after the one with the define or macro, regardless of the library where it was compiled.

If the design uses group mapping and has a Verilog config file, the tool uses the order specified in that file. If you do not have a Verilog config file, use `run compile -hdl_define __USELIBGROUPORDER__` or `__SYN_COMPATIBLE_INCLUDEPATH__` to specify search order.

## **-hdl\_define Macros**

With the standard compiler flow, you can define the following macros with the `-hdl_define` argument to the `run compile` or `run pre_instrument` commands:

|                                                                |                                                                                                                                                                                                                                                                                       |
|----------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__ISOLATE_LIB__</code><br>or<br><code>ISOLATE_LIB</code> | Controls accessibility and prevents inflow into a library.<br>See the preceding figure for a definition of inflow.<br>See <a href="#">__ISOLATE_LIB__, on page 117</a> .                                                                                                              |
| <code>SYN_COMPATIBLE</code>                                    | Ensures compatibility between different Synopsys tools such as Design Compiler (DC). DC ignores dynamic initialization assignments, unlike the prototyping tool. Use this macro to make DC compatible with the prototyping tool.<br>See <a href="#">SYN_COMPATIBLE, on page 117</a> . |
| <code>__SYN_COMPATIBLE_INCLUDEPATH__</code>                    | Specifies that the search path order for includes to be the same as the one used by the simulation tool (VCS), instead of the default search order.<br>See <a href="#">__SYN_COMPATIBLE_INCLUDEPATH__, on page 118</a> .                                                              |
| <code>SYN_STRICT_MODPORTS__</code>                             | Requires that modports defined strictly access the associated interface ports specified in the instantiation.<br>See <a href="#">__SYN_STRICT_MODPORTS__, on page 119</a> .                                                                                                           |
| <code>__USELIBGROUPORDER__</code>                              | Resolves conflicts when there are multiple module definitions in different libraries by using the file search order listed in the lmf file.<br>See <a href="#">__USELIBGROUPORDER__, on page 119</a> .                                                                                |

## \_\_ISOLATE\_LIB\_\_

By default, user-defined macros are accessible from HDL files that are defined in the library map where the macro was defined, but not from any other library map. If there is a group definition in the HDL, all macros defined for that group are constrained to that group.

Use \_\_ISOLATE\_LIB\_\_ to restrict macro search to the group. If this option is not defined, and there is no define for the group, the search looks for the macro in the global space.

If there is a define or macro specified in the library, this option prevents inflow access to that library. To prevent inflow for a library that does not contain a define or macro, use \_\_ISOLATE\_LIB\_\_/ISOLATE\_LIB to create a dummy define.

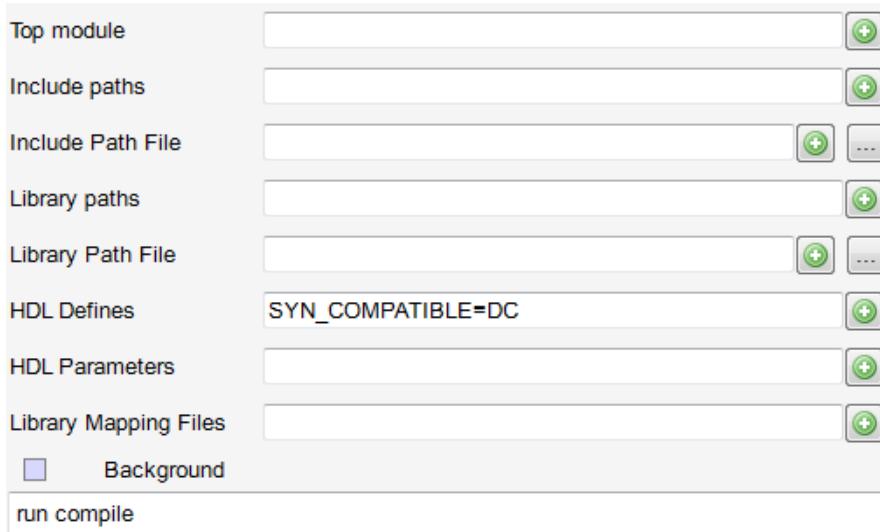
## **SYN\_COMPATIBLE**

Some Synopsys tools, such as Design Compiler (DC), ignore dynamic initialization assignments, unlike the prototyping tool. In the following example, note the line logic a=b. DC leaves the output unconnected, because it does not handle inline assignments. By contrast, the prototyping tool handles inline assignments, so input b drives output q.

```
module test (b, q);
 input b;
 output q;
 logic a = b;
 assign q = a;
endmodule
```

The tool warns you of the difference in handling (warning message @W: CG879), and treats the assignment as a regular assign. If you are using modules from DC and need them to be compatible with the prototyping tool, specify the following macro with a Tcl command or from the GUI:

1. To specify the macro with a Tcl command, use the `-hdl_define` argument to either the `run compile` or the `run pre_instrument` command:
  - `run compile -hdl_define SYN_COMPATIBLE=DC`
  - `run pre_instrument -hdl_define SYN_COMPATIBLE=DC`
2. To specify the macro through the GUI, do the following.
  - Go to the State->Run compile or Run pre-instrument dialog box.
  - In the HDL Defines field, set `SYN_COMPATIBLE=DC`

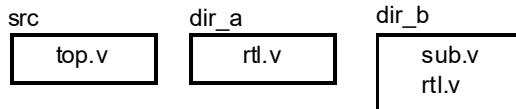


### \_\_SYN\_COMPATIBLE\_INCLUDEPATH\_\_

The default search order searches the current logical library of the file where the module is instantiated first, then the library path for the current logical library, and lastly other logical libraries.

With \_\_SYN\_COMPATIBLE\_INCLUDEPATH\_\_, the search path order is the same as VCS, where first priority with include file directories is given to RTL includes, and include path priority is lower.

With the default include path order of dir\_a then dir\_b, in the following example, top.v will include rtl.v from dir\_a and sub.v from dir\_b. Use \_\_SYN\_COMPATIBLE\_INCLUDEPATH\_\_ to pick up rt.v from dir\_b (RTL include) instead.



## \_\_SYN\_STRICT\_MODPORTS\_\_

Use the \_\_SYN\_STRICT\_MODPORTS\_\_ macro to require that modports defined strictly access the associated interface ports specified for the instantiation. You might encounter the following error:

**@E: CS172 The number of ports in the instantiation does not match the number of ports in the module definition for instance *instanceName***

This can occur if you have specified a global interface instantiation for modports using the “.\*” syntax to access the ports. Set the \_\_SYN\_STRICT\_MODPORTS\_\_ macro to ensure mapping completes successfully.

1. To specify the macro with a Tcl command, use the `-hdl_define` argument to either the `run compile` or the `run pre_instrument` command:
  - `run compile -hdl_define __SYN_STRICT_MODPORTS__`
  - `run pre_instrument -hdl_define __SYN_STRICT_MODPORTS__`
2. To specify the macro through the GUI, do the following.
  - Go to the State->Run compile or Run pre-instrument dialog box.
  - In the HDL Defines field, set \_\_SYN\_STRICT\_MODPORTS\_\_

## \_\_USELIBGROUPORDER\_\_

Use this directive to resolve conflicts when a module `myModule` is defined for multiple libraries. For example, if you specify this option when `myModule` is defined in both `lib1` and `lib2`, and `lib2` is defined before `lib1` in the `lmp` file, the tool uses the module definition from `lib2`.

If \_\_DefaultLibraryMapping\_\_ is also specified for the library in the `lmp` file, this macro uses the `run compile -srclist` file order as the search order for the libraries referenced by \_\_DefaultLibraryMapping\_\_.

If you also use Verilog configuration blocks to define precedence for conflict resolution, \_\_USELIBGROUPORDER\_\_ has a lower priority than the configuration blocks. It only comes into effect if a module is not resolved after applying configurations.

# Using Unified Compile

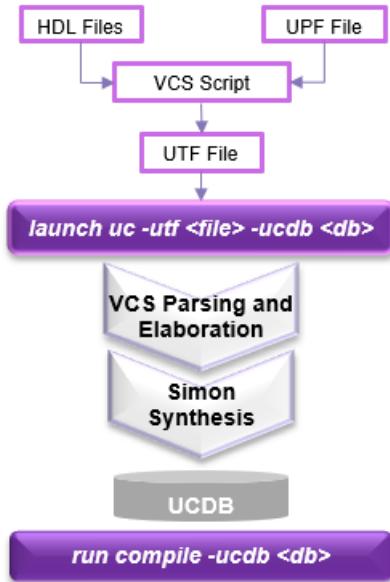
Use the Unified Compile flow to reuse VCS scripts and options, and to seamlessly move from the VCS environment to FPGA synthesis and prototyping. Semantics, language constructs, and library resolution are all consistent with the VCS software.

See the following for information on running the UC flow, and for recompiling incrementally after small design changes. Unless otherwise stated, these topics focus on the default UC 2.0 flow, not on older implementations of this functionality.

- [Unified Compile Design Flow](#), on page 120
- [Preparing the Input for Unified Compile](#), on page 125
- [Converting Designs for Unified Compile](#), on page 920
- [Running Unified Compile](#), on page 121
- [Checking the UC Database](#), on page 135
- [Compiling Incrementally Using Bypass Flow \(UC\)](#), on page 146
- [Compiling Incrementally Using Structural VM \(UC\)](#), on page 156
- [Running Bottom-Up Compile \(UC\)](#), on page 159

## Unified Compile Design Flow

The following figure summarizes the compile stage of the UC design flow. Subsequent stages after this follow the regular design flow. The figure shows the two main compile steps, where word-level VCS synthesis run first to create a UC database (ucdb), followed by another compile command on the compiled UC database.



For step-by-step guidance on the commands and this flow, refer to these topics:

- [Preparing the Input for Unified Compile](#), on page 125
- [Running Unified Compile](#), on page 121

## Running Unified Compile

The basic procedure below describes how to work with designs that might have been originally developed for other Synopsys verification tools and run Unified Compile from the FPGA prototyping tool.

1. Set up the design to run unified compile.
  - Follow the guidelines in [Preparing the Input for Unified Compile](#), on page 125.
  - Make sure to use compatible versions of the VCS and synthesis tools. Check the release notes for details.
  - Start with a VCS script. of the *Reference Manual*

- Follow the steps in [Converting Older Designs for Unified Compiler, on page 920](#) to use your prototyping design with the Unified Compile flow.
  - If you have IP, follow the recommendations in [Including IPs in a UC Design, on page 130](#).
  - For UPF information, create a UPF file and include it with the `vcs` command as described in [Setting up the UTF File and the VCS Script, on page 128](#).
2. Create a UTF file that points to the VCS script for compilation.
- The UTF (Unified Tcl Format) file is a common file for unified compile, where you can specify the design information. Not all UTF commands are supported. Check that the main command (`vcs_exec_command`) covers both the analyze and elaboration steps.
3. Start the unified compile process.
- Start the tool, and create a database. Make sure to specify the technology, which can vary. For example, use HAPS-80 for HAPS-80 designs or HAPS-1004F for HAPS-100 designs.
- ```
database load uc_debug -autocreate -technology technology
```
- If you are working with an existing database, load it with the `database load` command.
- To use formal verification, enable verification mode: option set `verification_mode 1`.
 - Launch the unified compiler from the Tcl window.
- ```
launch uc -utf path_to_utf_file -ucdb path_to_uc_database
```
- For details about the command syntax, refer to the corresponding descriptions in the [Command Reference](#). The `launch` command uses the VCS software to generate a netlist in the UC flow. After the VCS software elaboration stage, the word-level synthesis tool generates an intermediate netlist, which is then processed by the synthesis software.
- The default UC version is 2.0. You can specify version 1.0 with the `-v` option to the command, but it is not recommended.
4. Check the log file for errors and ensure that the results are clean.

The uc.log file is generated by the VCS software and is located in the directory specified for -ucdb. For example: ucdb/uc.log. The log file contains the reports generated by the VCS software. You can also check for UPF errors at this point. This is an excerpt:

```
=====
VCS pre-synth CPU Time : 0.28 sec
VCS pre-synth Wall-Clock Time : 1.00 sec
VCS pre-synth Current Memory : 307 MB
SIMON Master CPU Time : 0.02 sec
SIMON Total CPU Time : 0.02 sec
SIMON Master Wall-Clock Time : 0.15 sec
SIMON Master Current Memory : 309 MB
SIMON Master Peak Memory : 309 MB
SIMON CGroup Memory : 391 MB
=====
Info: Simon VCS Finished
Info: Simon call complete
Info: Exiting after Simon Analysis
CPU time: .360 seconds to compile
```

If you get an error about previously declared modules (Error-[MPD] Module previously declared), you have two options. Either fix the duplicate modules as described in [Setting up the RTL Design Files and Library Mapping, on page 127](#), or include the -error=noMPD option at the vcs command line to downgrade the error to a warning and allow duplicate modules.

5. Compile the database generated by the unified compiler.
  - Specify synthesis attributes as comments in the RTL or in a CDC file.
  - Set options in a Tcl options file. Most options, like resource sharing and retiming, are optional but the technology option is required. For encrypted designs, the result\_format vm option is also required. Source the options file.
  - Compile the design:

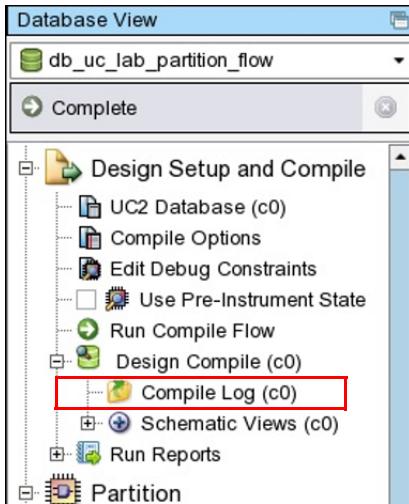
```
run compile -ucdb pathUCdatabase -cdc cdcFilePath
```

The process runs in distributed compile mode, ignoring any specified setting for parallel processing. If the design cannot be divided into parallel processes for distributed compile, the compiler runs in monolithic mode and prints the following message:

```
@N: This design does not have sufficient nodes/jobs for distribution, running in monolithic mode.
```

6. Check the results.

- Check the Unified Compiler Report section of the log file generated after run compile. Use the view report -state c0 command to open the report from the command line. From the GUI, go to the Database View panel and click on Compile Log.



- Run the report ctc command and check the log file it generates for clock information summaries.
- You can also use the VC Static tool to check for CDC violations and run lint checks (see [Running Lint Checks with VC Static, on page 137](#) and [Checking for CDC Violations with VC Static, on page 139](#)).

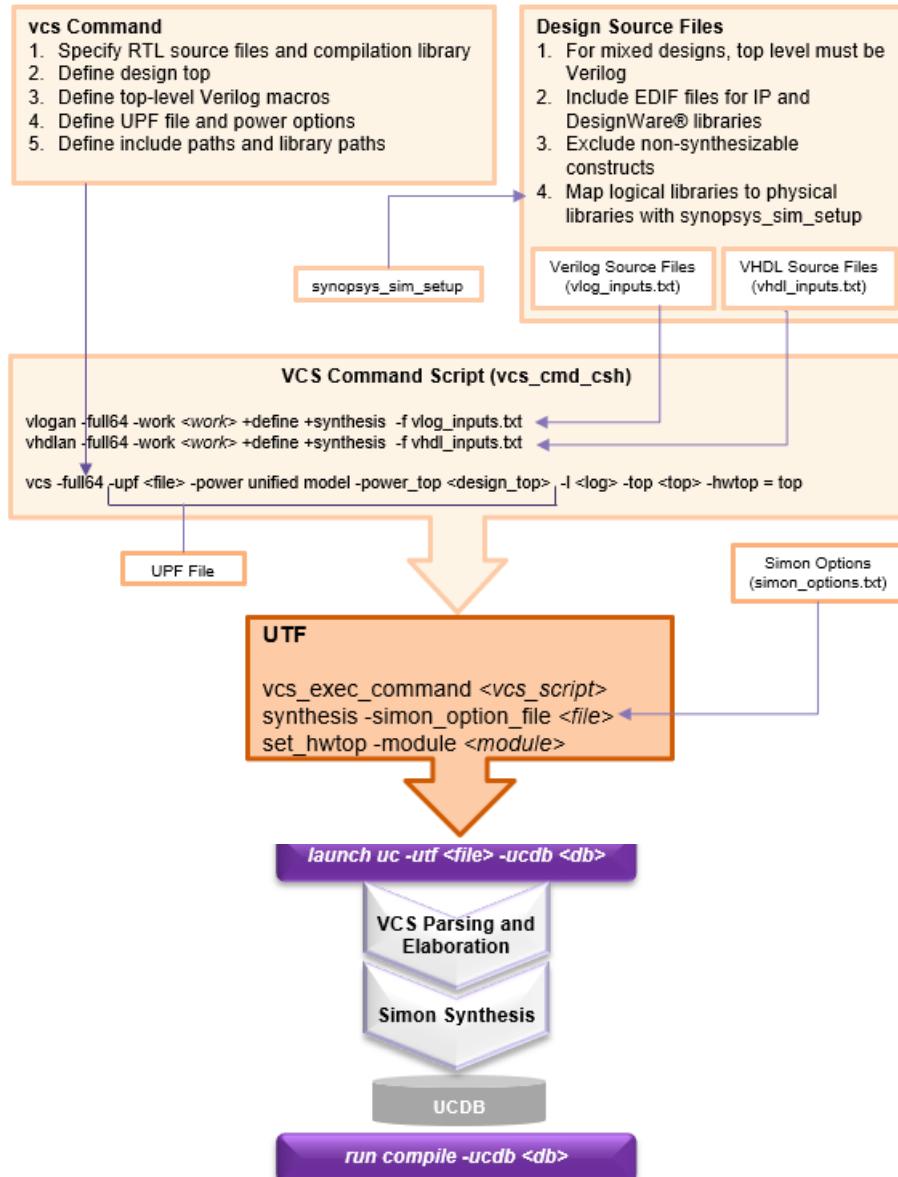
7. Continue with the rest of the flow as usual.

To run the single-FPGA synthesis flow, continue with the run pre\_map and run map commands.

To run the partition flow, continue with the run pre\_partition and run partition and other commands.

# Preparing the Input for Unified Compile

The following figure summarizes how the UC input information is structured so it can be specified for the launch uc command.



## UC Input Files and Environment Variables

This is a summary list of the UC input files shown in the previous figure; for details of setting them up, refer to [Preparation for the UC Flow, on page 126](#).

- HDL files: Verilog, SystemVerilog, or VHDL; or a mixture
- UTF file (points to the VCS script file)
- VCS script file (with or without pointer to UPF)
- UPF file (power information)
- Simon option file (Simon timing engine options)
- Synopsys\_sim.setup file (library mapping)
- Tcl script (includes the database load, launch uc, and run compile commands to run UC in the prototyping tool)

Set up environment variables to point to the installations for these tools:

- VCS installation path (UC)
- ProtoCompiler path (prototyping and FPGA synthesis)
- Xilinx Vivado path (place and route)
- Verdi installation path (verification)
- Design Compiler installation path (IP)
- Formality installation path (zFmCheck)

## Preparation for the UC Flow

The guidelines below provide some preparation tips to set up the files for the initial VCS front-end compile. Consult the *VCS User Guide* for further information.

- [Setting up the RTL Design Files and Library Mapping, on page 127](#)
- [Setting up the UTF File and the VCS Script, on page 128](#)
- [Including IPs in a UC Design, on page 130](#)
- [Including Complex SystemVerilog Port Modules, on page 133](#)
- [Handling HDL Differences, on page 921](#)
- [Converting Older Designs for Unified Compiler, on page 920](#)

## Setting up the RTL Design Files and Library Mapping

This procedure provides some guidelines on setting up the design source files and for mapping libraries; it is not exhaustive. The example snippets mainly show Verilog syntax; for the equivalent VHDL syntax and further details about the information here, refer to the *VCS User Guide*.

1. Use design source files that are in Verilog or VHDL formats, or a mixture of these languages.
  - Make sure the top level of the design is the synthesizable DUT, not the test bench.
  - For mixed designs, the top-level module in the design must be Verilog. If you have a VHDL top-level entity, create a Verilog wrapper for it and then specify the top level as usual (see step 4).
2. If you have modules that are included multiple times, use the recommended structure below to avoid compile time errors.

Define a macro which includes a declaration of the module:

```
`ifndef sample_mac
`define sample_mac
module myModule;
endmodule
`endif
```

The tool uses the declaration of myModule for multiple instantiations.

3. Exclude non-synthesizable constructs using one of the methods below:
  - Enclose non-synthesizable constructs inside translate\_off/translate\_on pragmas, using either of the syntax styles shown below:

```
/* synthesis translate_off */
...
/* synthesis translate_on */
```

or

```
/* synopsys translate_off */
...
/* synopsys translate_on */
```

- Add -skip\_translate\_body to vlogan/vcs to skip contents inside translate\_off/translate\_on. Use the same option with vhdlan to skip contents in VHDL designs.

4. Include IPs and other components by following the guidelines in [Including IPs in a UC Design, on page 130](#).
5. To add instrumentation, use SystemVerilog assertions (SVA), and the \$dumpvars (SystemVerilog) or \$dumports (VHDL) constructs.
6. Map logical libraries to physical libraries using the `synopsys_sim.setup` file.

The VCS software allows multiple logical libraries, each of which can point to a different physical library. The file syntax to map a logical library to a physical library is shown below. Logical library names are case-insensitive.

```
logical_name: physical_name
```

This is an example of the file. The first two lines specify that the default logical library, WORK, is mapped to the default physical library, work. The remaining lines specify the physical mapping for other logical directories and the DesignWare IP libraries to be added to the setup file:

```
WORK > DEFAULT
DEFAULT: ./work
L1 : ./lib/lib1
L2 : ./lib/lib2
L3 : ./lib/lib3
Lw1 : ./lib/lw1
Lw2 : ./lib/lw2
dw01 : ./dw01
dw02 : ./dw02
dw03 : ./dw03
dw04 : ./dw04
dw05 : ./dw05
dw06 : ./dw06
dware : ./dware
gtech : ./gtech
```

## Setting up the UTF File and the VCS Script

The UTF (Unified Tcl Format) file is an input file that specifies the design information for unified compile. in the *Reference Manual* it must point to the VCS script. Create a VCS script or specify the individual commands. The essential command in the UTF file is `vcs_exec_command`.

For further details about the script and the UTF file, refer to the VCS documentation.

1. Create a UTF file that points to the VCS script for compilation.

The required command is `vcs_exec_command`.

2. Set up the VCS script.

- Specify the required command, `vcs_exec_command`. For example:  
`vcs_exec_command {vcs_dut.csh}`
- Include the mandatory `vlogan` and `vhdlan` analysis commands, according to the HDL used for the design. The `vlogan` and `vhdlan` commands must be separated in the file. For information about the VHDL commands and any details about the VCS flow, refer to the *VCS User Guide*, which contains both Verilog and VHDL examples and detailed descriptions.
- Include the required `vcs` elaboration command.
- Include other VCS script commands that are optional. Some commonly used commands are described in the following steps.
- To get information about VCS script syntax, use `vcs -full64 -doc`.

3. of the *Reference Manual*Specify the top-level module in the VCS script.

- Specify the synthesizable design under test (DUT) as the top-level for analysis and elaboration in the VCS script.
- Define the top module with the `-hw_top` option.

`-hw_top= designUnderTest`

The `hw_top` model should be the top module of the synthesizable RTL. This option specifies that only the design below the specified module is to be elaborated.

#### Examples:

```
vlogan -full64 -sverilog sub_ip1.v
vlogan -full64 -sverilog sub_ip2.v
vlogan -full64 -sverilog dut.v
vcs -full64 -sverilog dut -hw_top = dut
```

or

```
vcs -full64 -sverilog sub_ip1.v sub_ip1.v dut.v -top dut
-hw_top=dut
```

- For synthesis and prototyping, exclude the testbench and transactors from the design setup. If the DUT has cross-module references (XMRs) from the testbench or transactors to the DUT, exclude them as well.
4. Explicitly define macros in the VCS script.

You cannot have implicit macros in the unified compile flow, so you must explicitly define any predefined macros. For example, if you have any synthesis or SYNTHESIS macros, they must be explicitly defined:

```
vlogan -full64 +define+SYNTHESIS +define+synthesis -sverilog
ip_1.v
```

5. Specify memory initialization files with \$readmemb/\$readmemh, and specify the associated include paths with the UTF `memory_preferences -scan_path` command.
6. To include low power information, do the following:
- Model low-power behavior in a UPF (Unified Power Format) file. in the *Reference Manual*
  - Add the `-upf` option to the VCS script to specify the UPF file. For example:  
`vcs -full64 top -hw_top = top -upf designName.upf`  
in the *Command Reference*For the complete syntax, refer to the VCS documentation.
  - If you have multiple UPF files, use one UPF file to source the others. VCS scripts only accept one UPF.

## Including IPs in a UC Design

A UC design can include IP and other components. Different components are described in the following sections:

- [Including Xilinx XPMs](#), on page 131
- [Including EDIF or NGC Files in a UC Database](#), on page 131
- [Adding Encrypted IP to a Unified Compile Database](#), on page 132

For further information, including encryption support, refer to the *VCS User Guide*.

## Including Xilinx XPMs

Including Xilinx Parameterized Macros (XPM) into a design enables the tool to read pre-defined RTL codes automatically and add the XPM definitions directly into the design during synthesis. It improves timing for the design and provides better synthesis results.

To use Xilinx XPMs in the UC flow, follow these steps:

1. Set the `XILINX_VIVADO` environment variable to a valid Vivado installation location.
2. Define the XPM library mapping in the `synopsys_sim.setup` file.  
`xpm : ./xpm`
3. Add the following code in the VCS script, to compile the SystemVerilog XPM files.

```
Create work library
mkdir xpm

Compile XPM Files in VCS
vlogan -work xpm
$XILINX_VIVADO/data/ip/xpm/xpm_memory/hdl/xpm_memory.sv
vlogan -work xpm
$XILINX_VIVADO/data/ip/xpm/xpm_fifo/hdl/xpm_fifo.sv
vlogan -work xpm $XILINX_VIVADO/data/ip/xpm/xpm_cdc/hdl/xpm_cdc.sv
```

## Including EDIF or NGC Files in a UC Database

Although the synthesis tool can directly read EDIF files, they must be handled differently when using unified compile because the VCS software cannot parse these files. You can either include the files with the `-srcList` argument, or use the bottom-up approach.

### Including EDIF or NGC Using `-srcList`

1. Create black box modules for the IP.

Specify the black box definitions when you compile the design. Link the modules into the compile state as described in the next two steps.

2. Create a text file that contains the files to be added.

List the EDIF or NGC files in the text file, using this syntax:

```
-type edif adder.edf
-type ngc adder.ngc
```

- When you compile, use the `-srclist` argument and specify the text file you created in the previous step:

```
database create db0
launch uc -utf run(utf -ucdb ucdb
run compile -ucdb ucdb -srclist EDIFfiles.txt
```

## Including EDIF or NGC Using the Bottom-Up Flow

- Compile the EDIF and NGC into a separate database, using the `run compile` command.

Use the `-srclist` argument with `run compile` to list the EDIF and NGC files. The EDF and NGC files can be listed in the same text file; use `-type` to distinguish the kind of files as described in [Including EDIF or NGC Using -srcList, on page 131](#).

- For the rest of the design, launch UC and create the UC database as usual.
- At the `run compile` stage, specify the `-src` option to “stitch” or incorporate the EDIF or NGC into the UC database.

This is an example of a typical sequence of commands, where `db0` is the EDIF database that is compiled separately and then added to the UC database, `db1`. The `add_files.txt` file contains the `-type` list of EDF/NGC files to add.

```
database load db0 -autocreate
run compile -srclist add_files.txt -top_module adder -out add
database load db1 -autocreate
option set design_flow synthesis
launch uc -utf run(utf -ucdb ucdb
run compile -ucdb ucdb -type state -src db0|add
```

## Adding Encrypted IP to a Unified Compile Database

You can add encrypted IP to your design, if it is encrypted with the VCS key.

- Encrypt the IP with the VCS key.

For detailed information refer to the chapter on encryption in the *VCS User Guide*.

2. Add the RTL for the IP as with any other RTL file.
3. Run the UC flow, and the IP is treated as any other RTL.
4. Check the results.
  - Open the schematic viewer and check that the IP module is displayed. You should be able to view the details in the IP module.
  - If the IP module is not linked successfully and becomes a black box, review the ucdb/src/*wrapperFile* and the IP top-level file and check that the module interfaces are identical. Most linking errors are caused by a mismatch in the module interfaces.

## Including Complex SystemVerilog Port Modules

Use the method described here to create a wrapper for a complex port module that was synthesized with another Synopsys tools and instantiate it in the RTL design.

1. Compile the design using UC.
2. Use zFMcheck to generate a simple vector port wrapper:
  - Run the zFMcheck command from the *installDir*/zebu/bin directory.
  - To generate a wrapper based on the hierarchy instance name, use the following command, where -d specifies the UC database directory generated in step 1:

```
zFmCheck -gen_rtl_wrapper_hier hier_instance_name -o zfm workdir -d
ucdb/design/defaultGroup
```

- To generate a wrapper based on the module name, use this command:

```
zFmCheck -gen_rtl_wrapper_module module_name -o zfm workdir -d
ucdb/design/defaultGroup
```

For example:

```
$BUILD/zebu/bin/zFmCheck -d ucdb -gen_rtl_wrapper_module sub1
-o
 wrappermodule_sub1
$BUILD/zebu/bin/zFmCheck -d ucdb -gen_rtl_wrapper_module sub2
-o
 wrappermodule_sub2
```

### 3. Prepare the files and compile the design.

- In the RTL code, replace the original code for the instances (sub1 and sub2 in this example) with their top wrapper module names. The following examples show the wrapper file paths for complex port modules sub1 and sub2:

```
wrappermodule_sub1/gen_rtl_wrapper/sub1_wrapper.sv
wrappermodule_sub2/gen_rtl_wrapper/sub2_wrapper.sv
```

- In the UFT file, declare the wrappers as black boxes:

```
vcs_exec_command {vcs_dut_wrap.csh}
synthesis -generate_db_for_fmcheck {true}
synthesis -blackbox sub1_sub_wrapper
synthesis -blackbox sub2_sub_wrapper
```

- Compile the design with the launch uc and run compile commands, as usual.

```
launch uc -utf runwrap.utm -ucdb ucdb_wrap
run compile -ucdb ucdb_wrap -out c0_wrap
```

### 4. Link in the port modules.

- Compile the port modules using the native compiler:

```
run compile -srclist files_list.txt -top_module
sub1_sub_wrapper -out c_sub
run compile -srclist files_list.txt -top_module
sub2_sub_wrapper -out c_sub
```

- Link these compiled databases for the port modules to the compiled database from the end of step 5.

```
run compile -srcstate pcucdb_wrap|c0_wrap -srcstate
pcucdb_sub1|c_sub
-srcstate pcucdb_sub2|c_sub -top_module top -out c0_top
```

# Checking the UC Database

Besides the UC log file (ucDatabase/uc.log), there are other tools and techniques available to validate the UC database. See the following for details:

- [Running zFmCheck to Validate VCS Design Files](#), on page 135
- [Running Lint Checks with VC Static](#), on page 137
- [Checking for CDC Violations with VC Static](#), on page 139

## Running zFmCheck to Validate VCS Design Files

You can check the elaborated RTL from VCS against the Simon output netlist using either the simulation (VCS) or formal verification (Formality) tools. The functionality used for validation is called zFmCheck.

1. Add this command to the UTF file:

```
synthesis -generate_db_for_fmcheck true
```

This creates readable RTL inside the ucdb folder:  
ucdb/design/defaultGroup/src\_for\_fm/

2. From a terminal, run the zFmCheck command on this database.
  - To run Formality equivalence checking, use the -scm argument with the command:  
  
`zFmCheck -scm -d design_dir -o output_dir -l time_for_run -j parallel_jobs  
-t top_module`
  - To run VCS simulation, use the -scm\_sim argument with the command:  
  
`zFmCheck -scm -scm_sim -d design_dir -o output_dir -l time_for_run  
-j parallel_jobs -t top_module`

The command runs VCS simulation when formal verification fails.

- Use the zFmCheck -help command to get immediate information about the syntax.

For an overview of the command and some examples, see [zFmCheck Command](#), on page 441 in the *Command Reference*.

### 3. Check the failures.

- Check the log file generated after the run. Refer to the example at [zFmCheck Log File Example, on page 136](#).
- Check ucdb/design/defaultGroup/failure\_list/Readme.txt.
- Run simulation only on the failures.
- Run the command with Verdi to debug issues: zfmCheck -Verdi\_debug

## zFmCheck Log File Example

```
#####
#SCM Formality summary#####
EQUIVALENT: 5899 (97%)
DIFFERENT : 16 (0%)
UNKNOWN : 170 (3%)
 |--REF_SETUP_FAIL : 22
 |--IMP_SETUP_FAIL : 2
 |--UNMATCHED_POINT : 127
 |--MULTIPLE_DRIVEN : 4
 |--TIMEOUT : 15
BLACKBOX : 0 (0%)
UNVERIFIED: 0 (0%)
SKIPPED : 1 (0%)
 |--UNSUPPORTED_BY_FM : 1
 |--CONDITION_MISMATCH_FOR_EVENT_CONTROL_LIST : 1
CLUSTER : 0 (0%)

6086
#####
#Scm Simulation summary#####
Modules : 186
PASS : 140
FAIL : 7
ERROR/MISSING : 39
RUNTIME-ERROR : 0
```

## Running Lint Checks with VC Static

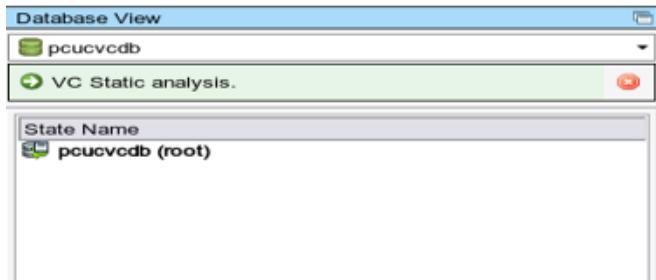
You can use VC Static functionality to run lint language and structural checks early in the design cycle.

1. Set up a VC Static configuration file (.txt), based on a sample file or a template.

See [VC Static Sample Configuration File, on page 141](#) for more about the contents of the file .

- Copy the sample file (vc\_static\_config.txt) from *install\_dir/lib/spyglass* into the working directory.
- Set the mode to PCLINT.
- The sample file points to a default template for lint mode. To customize the template (vcstatic\_lint\_template.tcl), copy the template from *install\_dir/lib/spyglass* to your working area, and customize it for your design. Do not modify the parts indicated in the template comments. Then point to the template you want to use from the sample file. For template format, refer to [Lint Template File, on page 143](#).

2. Open the database and specify the configuration file, before running compile.



- You can do this from the command line with the report vc\_static command:

```
report vc_static -utf myUTF.utf -path dir -config
vstatic_config.txt
```

- Alternatively, select Tools -> Report vc\_static and specify the configuration file from the GUI.

3. Run unified compile and compile, as usual.

4. Run pre-partition with VC static to generate a report.

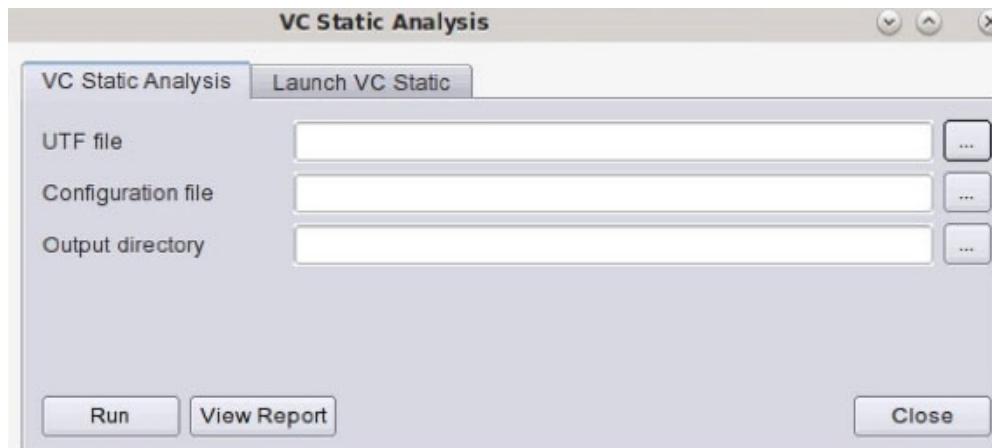
- From the command line:

```
run pre_partition -vc_static_cdc_path outputDir -tss file -fdc
file
```

- From the GUI:

Go to Tools -> Report vc\_static), specify the parameters, and click View Report. Note that the UTF file can be the same file as for unified compile, or can be a separate file.

The path for the output directory must be the same relative or absolute path specified with the report vc\_static command.



The tool runs static analysis and generates a cdc.pcf file with constraints for partitioning. The tool also applies the syn\_preserve 1, syn\_replicate 0, and syn\_allow\_retiming 0 attributes to all synchronization registers.

5. Export the results and analyze them:

- Use this command to export results:

```
export file -all -path outputDir
```

Check the partitioning constraints in the exported cdc.pcf file.

- Use the view reports command to display and check the log file, report\_lint.log.

## Checking for CDC Violations with VC Static

In larger designs where asynchronous clock networks are the norm, clock domain crossing (CDC) violations can be a problem. You can use VC Static functionality to identify synchronization logic early in the cycle and use it to set partitioning constraints that keep synchronization logic together.

1. Set up a VC Static configuration file (.txt), based on a sample file or a template.

See [VC Static Sample Configuration File, on page 141](#) for more about the contents of the file .

- Copy the sample file (vc\_static\_config.txt) from *install\_dir/lib/spyglass* into the working directory.
- Set the mode to PCCDC.
- The sample file points to a default template for CDC mode. To customize the template (vcstatic\_cdc\_template.tcl), copy the template from *install\_dir/lib/spyglass* to your working area, and customize it for your design. Do not modify the parts indicated in the template comments. Then point to the template you want to use from the sample file. For template format, refer to [CDC Template File, on page 142](#).

2. Open the database and specify the configuration file, before running compile.



- You can do this from the command line with the report vc\_static command:

```
report vc_static -utf myUTF.utf -path dir -config
vstatic_config.txt -show_report 1
```

- Alternatively, select Tools -> Report vc\_static and specify the configuration file from the GUI.

3. Run unified compile and compile, as usual.

4. Run pre-partition with VC static.

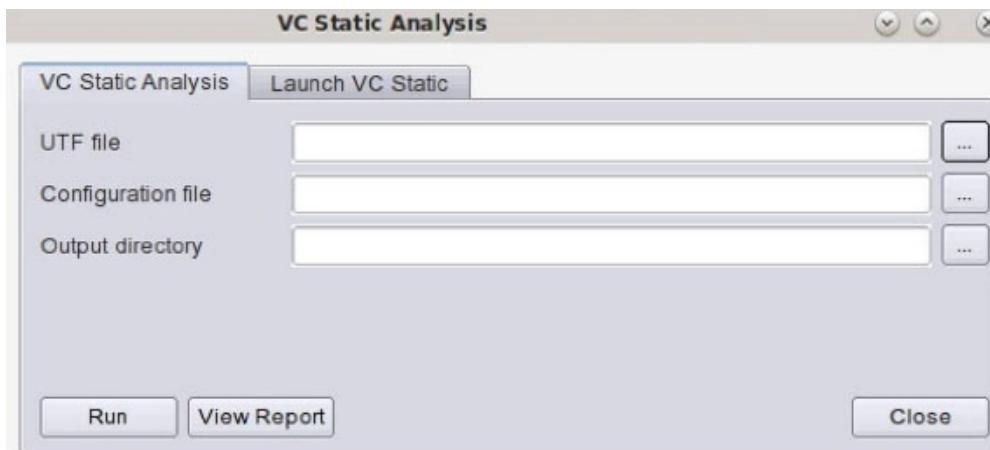
- To run it from the command line:

```
run pre_partition -vc_static_cdc_path outputDir -tss file
-fdc file
```

To run it from the GUI, go to Tools -> Report vc\_static, specify the parameters, and click View Report. Note that the UTF file can be the same file as for unified compile, or can be a separate file.

Initially, ensure to set the environment variable VC\_STATIC\_HOME to the relevant branch before invoking the VC Static feature. You must unset it for the next VC Static analysis.

The path for the output directory must be the same relative or absolute path specified with the report vc\_static command.



The equivalent Tcl command to run VC Static and load the CPDB database is launch vcstatic ([launch vcstatic, on page 75](#) in the *Command Reference*).

The tool runs static analysis and generates a generates a cdc.pcf file with constraints for partitioning. The tool also applies the syn\_preserve 1, syn\_replicate 0, and syn\_allow\_retiming 0 attributes to all synchronization registers.

- Export the results and analyze them:

```
export file -all -path outputDir
```

Check the log file and the partitioning constraints in the exported cdc.pcf file.

5. Use the constraints to guide partition.

## Example of Command Sequence for VC Static Integration

```
database load ucvcdb -autocreate -technology HAPS-80
report vc_static -utf uchaps.utf -path vcstatic -config
 vcstatic_config.txt -show_report 1
launch uc -utf uchaps.utf -ucdb ucdb
run compile -ucdb ucdb -out c0
run pre_partition -vc_static_cdc_path vstatic -tss board.tcl -fdc
top.fdc -out pp0
export file -all -reports_pp0
run partition ...
run system_route ...
run system_generate ...
```

## VC Static Sample Configuration File

The sample file is located in the *install\_dir/lib/spyglass* directory. The configuration file to run VC Static and identify clock domain crossing (CDC) violations must include these parameters:

|                                                                      |                                                                                                                                                                       |
|----------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Mode                                                                 | PCCDC, PCLINT, or both                                                                                                                                                |
| SDC (Synopsys Design Constraints)                                    | CDC flow. Design constraints must match the FDC constraints. You can enter FDC constraints here, but do not use FDC prefix notation, like n:, which is not supported. |
| Templates:<br>vcstatic_cdc_template.tcl<br>vcstatic_cdc_template.tcl | (Customization optional). Default templates are located in <i>install_dir/lib/spyglass</i> .                                                                          |

This is the sample file, vc\_static\_config.txt:

```
Choose PCCDC or PCLINT as the mode, or both separated by "|" (PCCDC|PCLINT).
MODE=PCCDC|PCLINT

Provide a list of SDC files separated by "|" (file1.sdc|file2.sdc).
SDC definitions should match exactly with the RTL definitions.
SDC=file1.sdc | /path1/file2.sdc

Enter user-defined cdc_template.tcl file and lint_template.tcl files.
The default templates are available in the <PC_BUILD>lib/spyglass/ directory.
You may copy these templates and prepare customized templates to run the flow.
If relative paths are used for the customized templates, make sure paths are relative to the current working di
If the following lines are commented out, templates in the default directory <PC_BUILD>lib/spyglass/ are used.

cdc_user_template=<cdc_user_template.tcl>
lint_user_template=<lint_user_template.tcl>
~
```

## CDC Template File

You can customize the template, but do not modify the parts indicated.

```
#####
DEFAULT CDC TEMPLATE FILE USED FOR VC_STATIC FLOW IN PROTOCOMPILER
#####

set session_name vccdc.daidir/run_vcsgcdc

Setting application variable to enable VCSGCDC-PC flow
set_app_var enable_cdc true
set_app_var read_synopsys_sim_setup true

#####
BELOW LINES SHOULD NOT BE MODIFIED
#####

Template file for "vcs -vccdc"
Assume read_command variable is set

RECORD:ON
eval $read_command
RECORD:OFF

eval $sdc_command
configure_cdc_nff_sync -detect_full_chain true
check_cdc

Report Generation
set log_path "."
if {[info exists ::env(SNPS INTERNAL PC CDC DIR)]} {
 set log_path $env(SNPS INTERNAL PC CDC DIR)
}

report_cdc -verbose -file $log_path/report_cdc.log
```

## Lint Template File

You can customize the template, but do not modify the parts indicated.

```
#####
DEFAULT LINT TEMPLATE FILE USED FOR VC_STATIC FLOW IN PROTOCOMPILER FLOW
set session_name vclint.daidir/run_vcsglint

Setting app var to enable VCSGLINT-ZEBU flow
set_app_var enable_lint true
set_app_var enable_zlint true
set_app_var read_synopsys_sim_setup true

#####
BELOW LINES SHOULD NOT BE MODIFIED
#####

Template file for "vcs -vcstatic"
Assume read_command, session_name variables are set

RECORD:ON
eval $read_command
eval $sdc command
RECORD:OFF

check_lint

set log_path "."
if {[info exists ::env(SNPS_INTERNAL_PC_LINT_DIR)]} {
 set log_path $env(SNPS_INTERNAL_PC_LINT_DIR)
 report_lint -limit 0 -verbose -include_waived -gen_empty -file $log_path/report_lint.log
} else {

 # Report Generation
 report_lint -limit 0 -verbose -include_waived -gen_empty
}
checkpoint_session -session $session_name
To view results, start vc_static_shell and execute:
restart_session -session $session_name
view_activity

quit
```

## Current Limitations

The VC Static integration currently has some limitations:

- Data and control synchronizers are not supported.
- Specify FDC constraints through the SDC parameter in the configuration file.
- FDC prefix notation (p:, c:, n:) is not supported.

# Compiling Incrementally

Given the current norm of large complex designs, minor design modifications could become very expensive time-wise if the design cannot be recompiled incrementally.

With a UC incremental run, the tool skips partitioning and only reruns the individual FPGAs that were changed, thus saving turnaround time. UC incremental runs differ from incremental runs with the standard compiler, because the UC flow does not have a system-level compiled database and must work from the single-FPGA compiled databases instead.

See these topics for information about running incremental compile in the UC and standard compiler flows:

- [Design Database to SRS Incremental Flow \(UC\)](#), on page 144
- [Compiling Incrementally Using Bypass Flow \(UC\)](#), on page 146
- [Compiling Incrementally Using Structural VM \(UC\)](#), on page 156
- [Using the Bypass Flow \(Standard Compiler\)](#), on page 165

## Design Database to SRS Incremental Flow (UC)

With Design Database to SRS incremental flow (SCM to SRS incremental flow), only modules that are modified for the next run are compiled. Designs or modules that are not modified are reused.

Use the -incremental option with the run compile command to do incremental runs.

You must use the same UCDB name for the first compile run and subsequent runs to make sure only the changed modules are recompiled. Make a copy of the UCDB before rewriting it for debugging.

### First Run

1. Run VCS.

```
launch uc -utf <UTF file path> -ucdb <Path to create design database>
```

2. Build the design

```
run compile -ucdb <Path to design database> -out <compile state name>
-incremental
```

### Example

```
launch uc -utf run.utf -ucdb ucdb
run compile -ucdb ucdb -out c0 -incremental
```

### Incremental Run

1. Build the design with changes in RTL.
2. Clone the compile state.

```
database clone_state -state c0 -name cincr
```

3. Run the unified compiler with the same database name used during the first run.

```
launch uc -utf run.utf -ucdb <database name used in first run> -v 2.0
```

4. Compile only the modified modules using the -incremental option with run compile command.

```
run compile -ucdb <Path to design database> -out <new compile state name>
-incremental
```

Example:

```
run compile -ucdb ucdb -out cincr -incremental
```

5. Check compile\_summary.rpt report to confirm the success of the incremental run.

The report is available in the following path:

<database name>\_reports/ <new compile state name>/compile\_summary.rpt

Example:

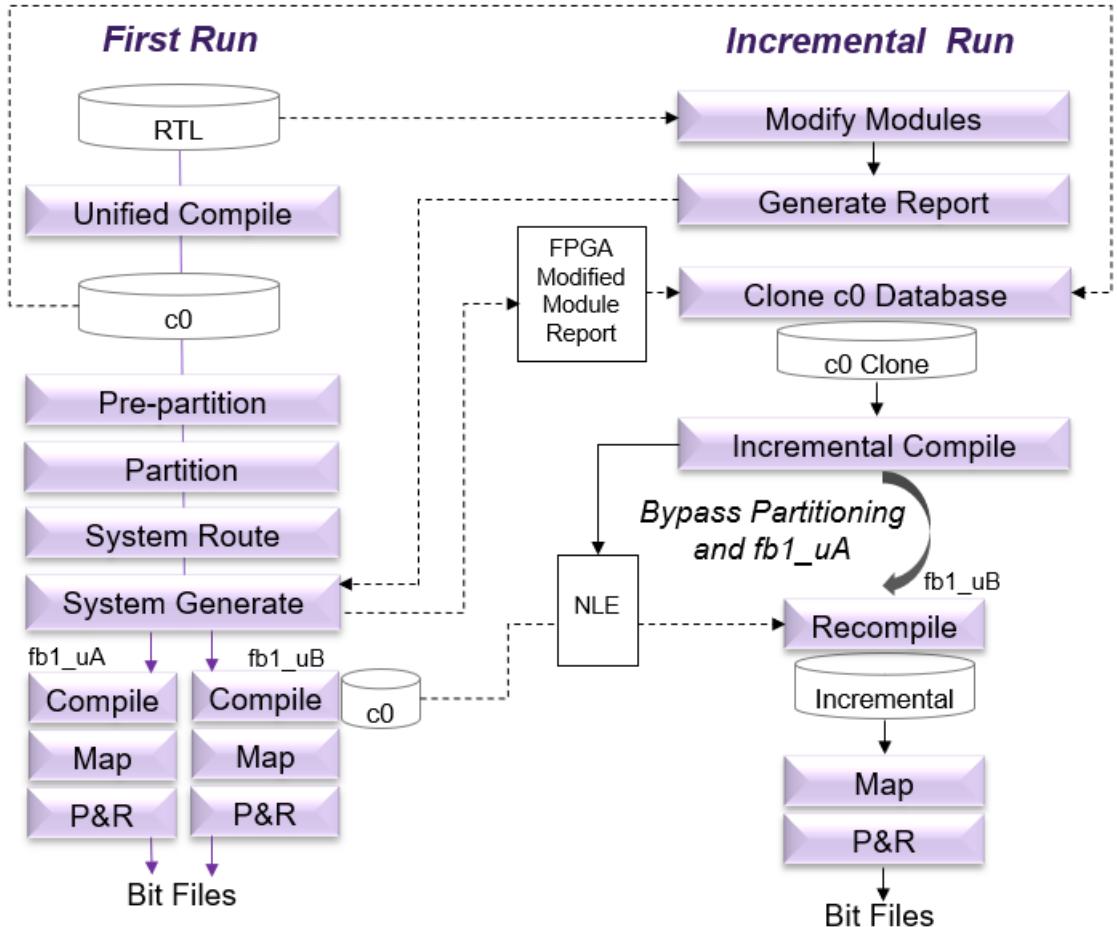
If the database name is ucdb and the new compile state name is cincr, then the report is ucdb\_reports/cincr/compile\_summary.rpt.

## Compiling Incrementally Using Bypass Flow (UC)

The bypass flow or SRS bypass flow is an incremental methodology in the UC flow, used for minor RTL changes at the design level within an FPGA. It bypasses partitioning and routing, and only recompiles the modified parts of the design in the affected FPGA. Updated views for the modules with changes are linked in to the rest of the design.

If you have modifications at the FPGA level, use the alternative method for compiling incrementally, described in [Compiling Incrementally Using Structural VM \(UC\), on page 156](#).

The following figure summarizes the UC bypass flow, showing both the initial and incremental runs. The black arrows connect the steps in the incremental flow.



You can use either the automated SRS bypass flow or the manual SRS bypass flow:

- [Automated Bypass Flow \(UC\), on page 148](#)
- [Manual Bypass Flow \(UC\), on page 152](#)

## Limitations

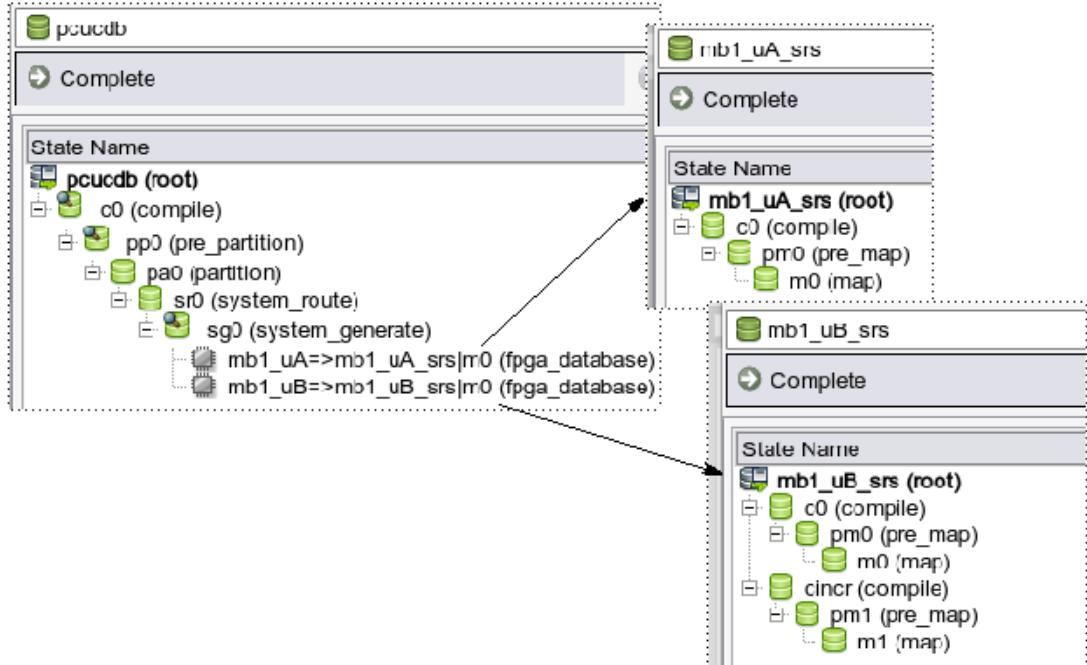
- Linking might fail in the following scenarios:
  - RTL changes that modifies ports of modules.

- XMR changes that impacts port changes.
  - Instrumentation changes that cause port changes.
  - Modules partitioned across FPGAs.
- Module names with special characters cannot be selected for SRS bypass flow. In such cases use its parent module while linking.
  - Modules impacted by partitioner is marked as Modified. These modules cannot be edited using SRS bypass flow.
  - The bypass flow reuses FDC. If modifications are affecting critical paths, you must manually update the appropriate FDC constraints.

## Automated Bypass Flow (UC)

With the automated SRS Bypass Flow, you need not find the module, where it is partitioned to, or run individual FPGA scripts manually. The tool gets all necessary NLEs and constraints applied to the design.

1. Make incremental changes in a design that has been run through the flow, through the place-and-route stage.
  - The top-level design would have gone through all the design states through system generate, and the individual FPGAs would have database nodes for compile, pre-map, and map. The next figure shows parent-level and FPGA-level database states for a design with two FPGAs.



- Make minor module-level design changes. The changes must be within a single FPGA.
- 2. Check the modules to be bypassed on the incremental run.

The bypass methodology can only be applied to modules that were not modified by the partitioner. Note that modules impacted by global optimization operations like constant propagation or sequential optimizations will be listed in the report, but these modules cannot be edited with the bypass flow. For these cases, use the structural VM flow instead. You can generate a report that lists the modified modules with the `export file` command.

- To export a list of modules, run `export file` for each FPGA from the top-level `system_generate` state, and export a report that lists them:

```
export file Modified_Modules_mb1_uA.rpt
export file Modified_Modules_mb1_uB.rpt
```

- To see a list of all modified files available, run this command from the `system_generate` state:

```
export file -list -all
```

3. Clone the compile state to create a duplicate state for incremental changes.

The main reason for cloning the state is control, so that you can run multiple modification iterations or revert to the pre-modified state if needed.

```
database load pcucdb
database set_state c0
database clone_state -name c0_clone
```

4. Launch incremental compilation and specify the original and clone states for `run compile`.

- Start from the root database of the previous run, and launch UC.
- Use the `run compile` command, as usual, but make sure to specify the clone compile state with `-out`. If there are multiple iterations with multiple clone states, make sure to point to the correct one.

```
database load pcucdb
database set_state root
launch uc -utf run.utf -ucdb ucdbincr
run compile -ucdb ucdbincr -gen_update_nle 1 -out c0_clone
```

The tool runs incrementally and recompiles the modified modules. The `run compile` command creates `nle` files with replacement commands for the module views with changes.

5. Export the modules with changes.

- Use the following command to export the modules:

```
export file update_modified_views.nle
```

This is an example of the NLE file. The NLE `replace_view` command replaces the whole sub-hierarchy it points to. There is a checksum for modules in `vcs.xcui`.

```
Generated by Synopsys Netlist Linker
version comp202003pcp2, Build 015R on Jun 5 2020 09:17:51
NLE Script to edit modified views for SRS Bypass Flow
set srs_handle [load_design \
 "myDesign/pcucdb/compile/c0_clone/synwork/top_comp.srs"]
replace_view {top_clk_inst.U3} $srs_handle:seqlog3
```

- Manually edit the nle file if needed.
- In the NLE, comment out any replacement views that are not part of that FPGA; if not, linking (next step) will fail.
- If you modify multiple modules that are targeted to different FPGAs, edit the NLE script manually to point the correct compiled (srs) handle for that FPGA.

## 6. Select system generate state

```
database set_state {sg0}
```

## 7. Check which post-partition FPGA is affected and export the generated bypass log.

```
report srs_bypass -compile_state c0_clone
export srsbypass.log
```

The report srs\_bypass command generates a report with details of the modules changed, post-partitioned FPGAs affect, and whether the incremental SRS bypass flow can be used. The srsbypass.log contains details of each module that has been changed.

## 8. Apply the changes to a single FPGA node.

```
export srs_bypass -compile_state c0_clone -path <path> -use_mapper
```

The -use\_mapper option finds the parent view if the child is in a non-incremental view. The tool traverses through the hierarchy till it finds a view which can be replaced.

The export srs\_bypass command exports the following files:

- sg0+cincr\_setup.tcl—Contains FPGA name that needs to be re-run after the update.
- sg0+cincr\_top.tcl—Calls individual FPGA scripts that are in the pcucdb\_synthesis\_files/mb1\_u\* folder.
- cincr\_update\_modified\_views.nle—Contains the replace view commands of updated module needed for stitching original SRS. This file saved in the pcucdb\_synthesis\_files/mb1\_u\* folder.
- cincr\_mb1\_uC\_srs.tcl—Calls FPGA script mb1\_uC\_srs.tcl to launch from beginning.
- report\_srs\_bypass.log—Log file stating whether the SRS bypass flow can be used. Lists the modules that are changed and provides details on

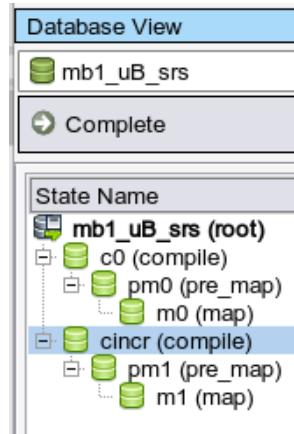
whether a module can be linked or not. The parent view names are also listed in this log.

9. Link the updated compiled SRS to individual FPGAs.

```
cd <path>
source sg0+c0_clone_top.tcl
```

10. Runs the whole FPGA with all constraints applied on compile, pre\_map, and map states. Any failure in linking can be debugged by checking the message in netlist\_optimizer.log.

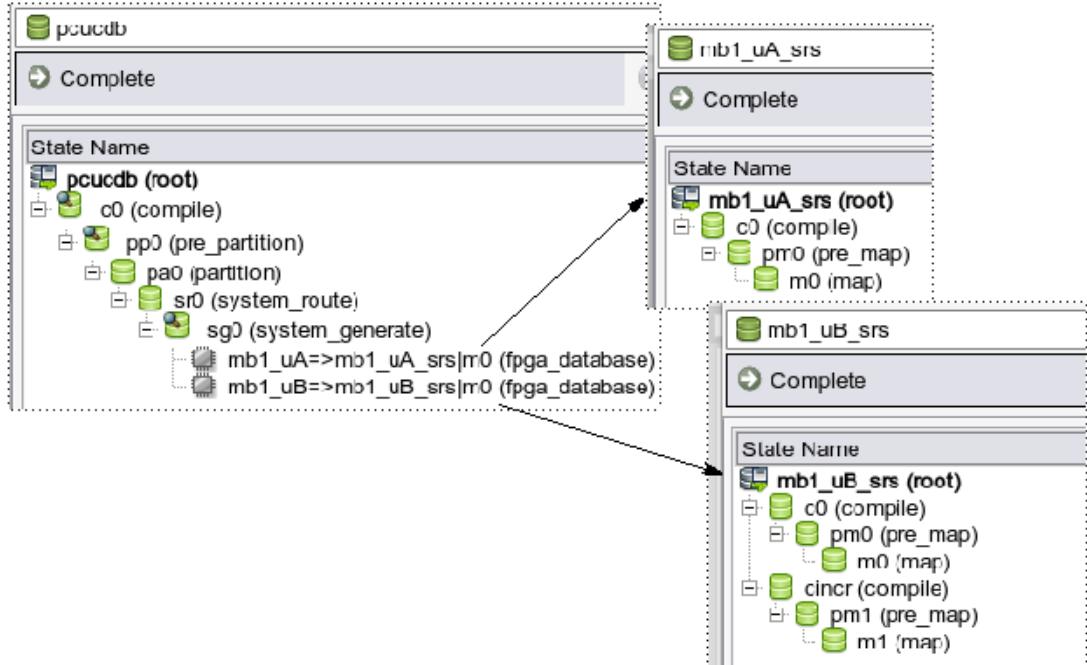
The following figure shows the results of successful linking:



Complete the incremental design by running the FPGA through the remaining design stages from map state to bit generation.

## Manual Bypass Flow (UC)

1. Make incremental changes in a design that has been run through the flow, through the place-and-route stage.
  - The top-level design would have gone through all the design states through system generate, and the individual FPGAs would have database nodes for compile, pre-map, and map. The next figure shows parent-level and FPGA-level database states for a design with two FPGAs.



- Make minor module-level design changes. The changes must be within a single FPGA.
- 2. Check the modules to be bypassed on the incremental run.

The bypass methodology can only be applied to modules that were not modified by the partitioner. Note that modules impacted by global optimization operations like constant propagation or sequential optimizations will be listed in the report, but these modules cannot be edited with the bypass flow. For these cases, use the structural VM flow instead. You can generate a report that lists the modified modules with the `export file` command.

- To export a list of modules, run `export file` for each FPGA from the top-level `system_generate` state, and export a report that lists them:

```
export file Modified_Modules_mb1_uA.rpt
export file Modified_Modules_mb1_uB.rpt
```

- To see a list of all modified files available, run this command from the `system_generate` state:

```
export file -list -all
```

3. Clone the compile state to create a duplicate state for incremental changes.

The main reason for cloning the state is control, so that you can run multiple modification iterations or revert to the pre-modified state if needed.

```
database load pcucdb
database set_state c0
database clone_state -name c0_clone
```

4. Launch incremental compilation and specify the original and clone states for `run compile`.

- Start from the root database of the previous run, and launch UC.
- Use the `run compile` command, as usual, but make sure to specify the clone compile state with `-out`. If there are multiple iterations with multiple clone states, make sure to point to the correct one.

```
database load pcucdb
database set_state root
launch uc -utf run.utf -ucdb ucdbincr
run compile -ucdb ucdbincr -gen_update_nle 1 -out c0_clone
```

The tool runs incrementally and recompiles the modified modules. The `run compile` command creates `nle` files with replacement commands for the module views with changes.

5. Export the modules with changes.

- Use the following command to export the modules:

```
export file update_modified_views.nle
```

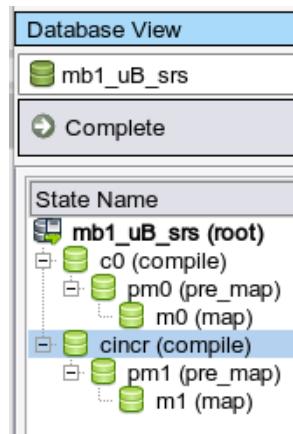
This is an example of the NLE file. The NLE `replace_view` command replaces the whole sub-hierarchy it points to. There is a checksum for modules in `vcs.xcui`.

```
Generated by Synopsys Netlist Linker
version comp202003pcp2, Build 015R on Jun 5 2020 09:17:51
NLE Script to edit modified views for SRS Bypass Flow
set srs_handle [load_design \
 "myDesign/pcucdb/compile/c0_clone/synwork/top_comp.srs"]
replace_view {top_clk_inst.U3} $srs_handle:seqlog3
```

- Manually edit the `nle` file if needed.
  - In the NLE, comment out any replacement views that are not part of that FPGA; if not, linking (next step) will fail.
  - If you modify multiple modules that are targeted to different FPGAs, edit the NLE script manually to point the correct compiled (`srs`) handle for that FPGA.
6. Link the modules with changes to the original compiled database by specifying the `nle` file:
- For each modified FPGA, recompile with `run compile`, and specify the `nle` file generated in step 5:

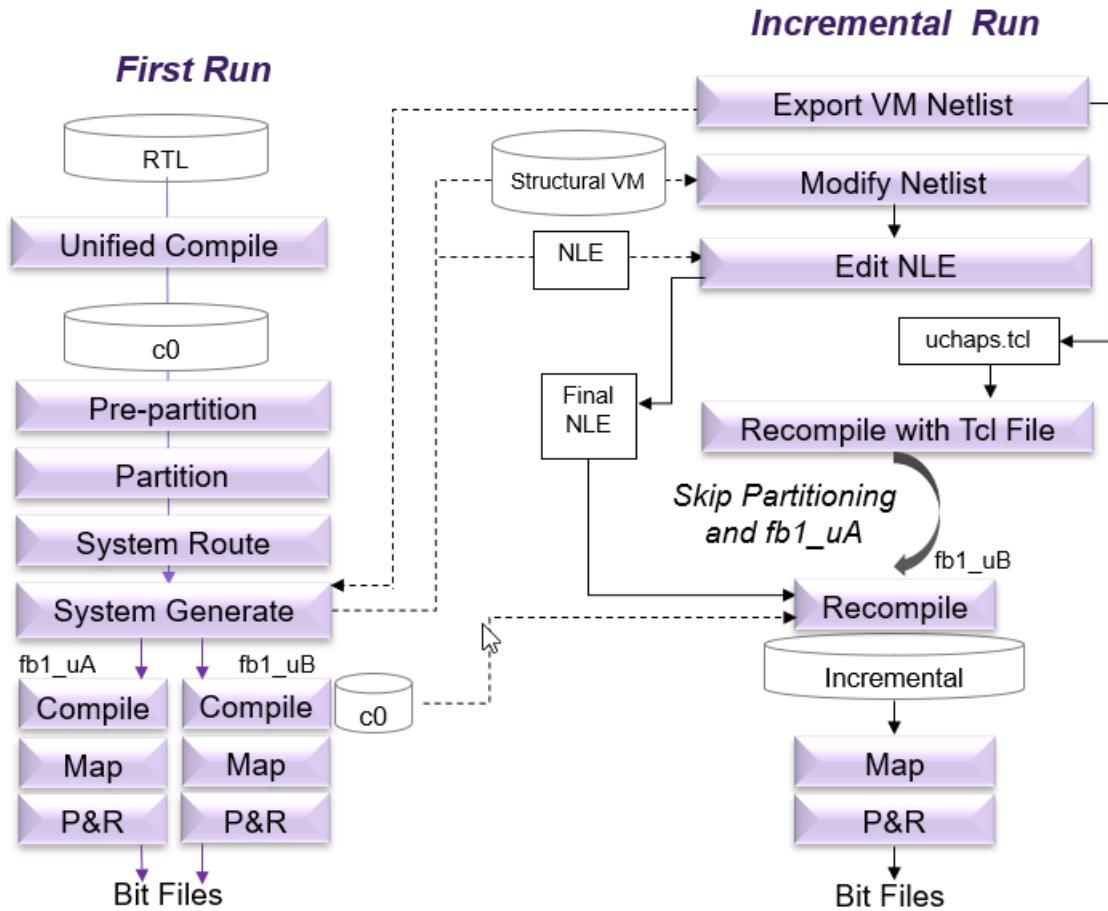
```
database load pcucdb_synthesis_files/mb1_uB/mb1_uB_srs
database set_state c0
run compile -srcstate c0 -nle update_modified_views.nle -out
cincr
```
  - You cannot recompile modules that have escaped identifiers as names . For these modules, use the module parent while linking.
  - If there are link failures, check messages in the `netlist_optimizer.log` file to debug it. Linking might fail because of these reasons: RTL changes that modify the ports of modules, XMR changes that cause port changes, changes to instrumentation that cause port changes, or modified modules partitioned across FPGAs.

The following figure shows the results of successful linking:



7. Complete the incremental design by running the FPGA through the remaining design stages from pre-map to bit generation.

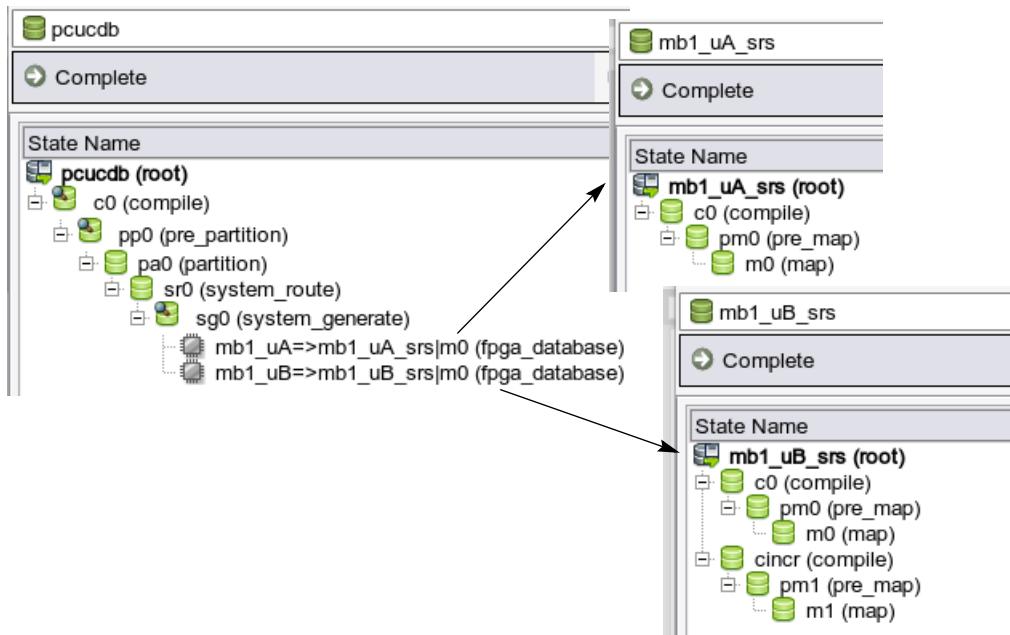
## Compiling Incrementally Using Structural VM (UC)



The structural VM flow is an incremental run methodology, for when changes are made at the FPGA level. A structural netlist is module-based instead of behavior-based. As the figure shows, the flow starts with the export of a structural netlist (.vm) from the System Generate stage. Changes are made in the structural netlist and linked back to the top-level design. This allows you to skip the partitioning stages at the top level and rerun only the FPGA with the

changes. Use this method for changes within an FPGA, or when the bypass method (*Compiling Incrementally Using the SRS Bypass Flow (UC), on page 157*) cannot be used.

1. Make changes to a design that has run through the flow, through place and route.
  - In a partitioned design, the top-level design would have gone through all the design states through system generate, and the individual FPGAs would have database nodes for compile, pre-map, and map. The figure shows parent-level and FPGA-level database states for a design with two FPGAs.



2. Export the structural VM netlist from the System Generate state, using the export netlist command:
  - To export a complete FPGA, use this command:
 

```
export netlist -path <OUT_DIR> -format structver
```
  - To export selected modules, use this syntax:
 

```
export netlist -path <OUT_DIR> -format structver
 -modlist <FILE_WITH_LIST_OF_MODULES>
```

The modules can be specified in the file in these ways:

```
i:<instanceName>
v:<cellName>
lib.cell.view.
```

Note that if you specify multiple modules, the VCS script is only created for the first module in the list. You must modify the script to compile multiple modules.

The `export` command creates the UC scripts required to compile the structural VM netlist. It generates the following files:

|                                        |                                                                   |
|----------------------------------------|-------------------------------------------------------------------|
| <code>nle_script</code>                | NLE file with replace view commands for exported module instances |
| <code>struct_ver</code>                | Structural VM netlist                                             |
| <code>system_ip</code>                 | System IP files                                                   |
| <code>uchaps.utf</code>                | UTF commands                                                      |
| <code>uchaps.tcl</code>                | UC compilation commands                                           |
| <code>update_modified_views.nle</code> | NLE script for entire FPGA                                        |
| <code>vcs_cmd.csh</code>               | VCS commands for UC compilation                                   |

3. To make incremental changes to modules that are modified by the partitioner follow these steps:

This flow does not support instrumentation that is added outside the DUT (design under test) instance in the structural netlist.

- Make minor changes to the VM netlists for the modules.
- Add the `-structver` option in the NLE file to force the linking of these modules. Use `-structver` the tool to accept netlist changes necessitated by modifications by the partitioner, like net name changes, logic optimizations, distribution across FPGAs etc.

```
replace_view -structver {u1} $srs_handle:module_pplan
```

4. Recompile the structural netlist by sourcing the `uchaps.tcl` file created by the `export` command.
5. Go to the FPGA with the changes and do the following:

- Recompile the FPGA and use the NLE file to link it to the old compiled database:

```
run compile -srcstate c0 -nle <NLE_file> -out cincr
```
- Continue with the rest of the flow for the FPGA as usual, through bit file generation.

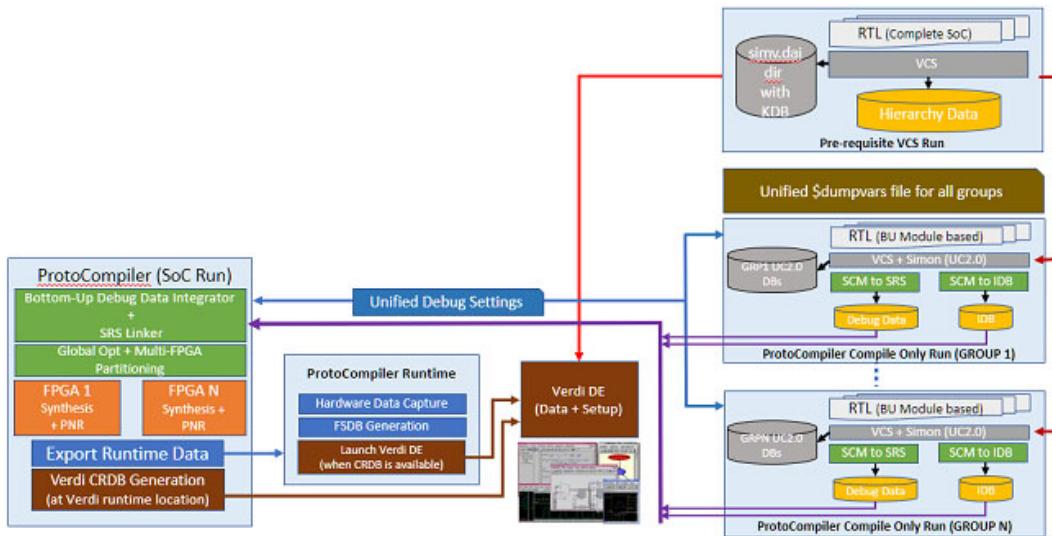
## Limitations

- When exporting multiple modules, the VCS script is created for the first module as the top module. Modify this script if you have to compile different modules.
- Modules that have escaped identifiers as names cannot be selected by the compiler for the structural VM flow.
- Debug logic that is outside the DUT will not be instrumented.

## Running Bottom-Up Compile (UC)

Bottom-up compilation is a useful technique for managing complex designs with divide-and-conquer strategies, to distribute and divide work between teams, to incorporate IP, or to isolate sections of designs.

Run bottom-up compile by first compiling individual blocks and then stitching them together and running compile at the top level. You can run bottom-up compilation on a single database where you stitch together multiple compile database states, or stitch together compiled database states from different databases.



## Prerequisites to Run the Flow

- Preparing the design
  - The entire design is run using the VCS flow without any black boxes meant for compilation groups.
  - Generated Verdi KDB and hierarchy database remains available.
  - The database is re-generated for all RTL changes.
  - Design hierarchy is exactly the same as the one created after running and merging all compilation groups.
- Running compilation groups
  - RTL partitioning is done with appropriate black boxing.
  - Matching sub-tree hierarchies are available within each group with respect to the entire design.
  - Each compilation group is run at separate location with same debug options.
  - Hierarchy information is generated from VCS pre-requisite run.

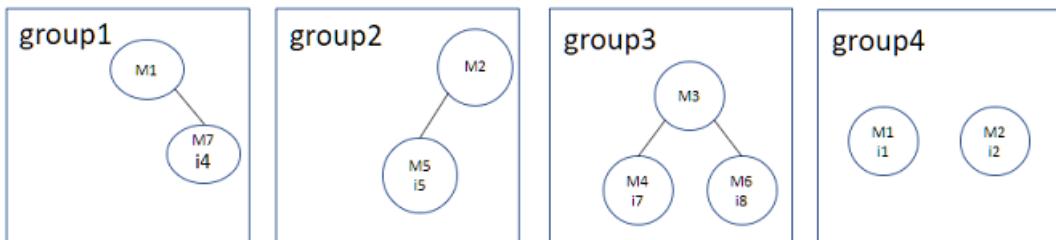
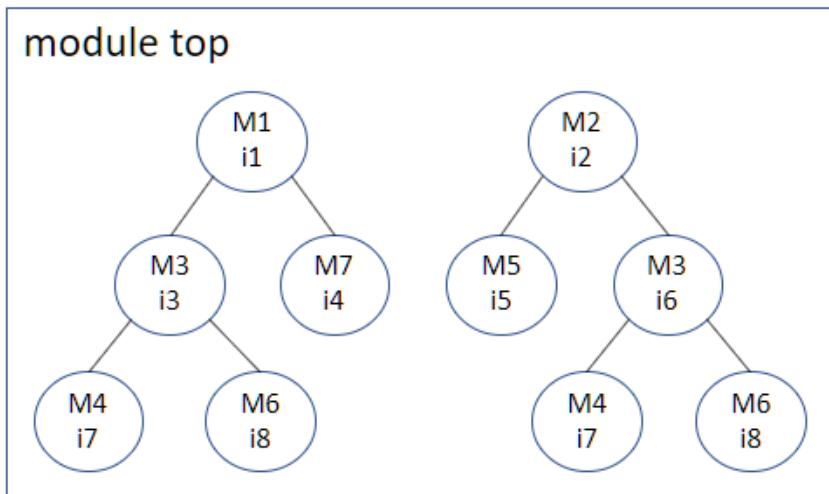
- RTL instrumentation is applied and recommended only through \$dumpvars, or SVA.
- Top module compilation group requires IDC specifying all the debug IP configurations, clock specification. The IDC instrumentation and clock specification should only be within the top scope.
- Debug visibility reports can be used within each compilation group.
- Running prototyping, and Runtime flow:
  - All compilation groups are provided for integration with appropriate top specification.
  - SRS-based IDC specification are applied at pre-partition after the compilation groups are integrated. The IDC specification can also be given during FPGA synthesis.
  - All the existing debug features are available after the compilation groups are integrated.
  - Verdi KDB generated from VCS pre-requisite run is used during the runtime for debug session.

To run the flow, see the following sections:

1. [Preparing the Design Using the VCS Software](#), on page 163
2. [Running Compile on Compilation Groups](#), on page 163
3. [Debugging with the Verdi Software](#), on page 164

## Understanding Compiler Groups

Using compiler groups you can define modules or part of a design hierarchy that needs to be run through unified compile (-compile\_only) flow independently.



### Unified \$Dumpvars Specification for the Flow

- VCS pre-requisite run for complete SoC
- Appropriate partitioning of the SoC
- Each Compilation group hierarchy is a sub-tree of SOC design hierarchy
- One specification is applicable across all groups
- Limit specification is not available
- Appropriate instrumentation is created based on path specifications
- Supports wild cards, dynamic force, and monitor specification

Example \$Dumpvars specification for the compile group based on the previous diagram:

```
$dumpvars(1, top.i1.i4)
$dumpvars(1, top.i2.i5)
$dumpvars(1, top.i1.i3.i7)
$dumpvars(1, top.i1.i3.i8)
$dumpvars(1, top.sig*)
```

### Unified SVA Specification for bottom-up flow

- SVAs can be written within any module
- Appropriate instrumentation is created for all module instantiations containing SVA

## Preparing the Design Using the VCS Software

1. Set the option **-hierarchy\_info=<path>** to generate the hierarchy information for each module.

```
export HIERARCHY_INFO=/bottom_up/pre_requisite_run
```

or

Provide the absolute path to the `hierarchy_info` argument directly.

```
/bottom_up/pre_requisite_run
```

2. Set the option **-kdb** for each VCS command to generate RTL Knowledge Database (KDB) to be used for Verdi debug. Note that for each RTL change, for any module, you must do a re-run to re-generate the RTL KDB.

```
/bottom_up/pre_requisite_run/run_vcs?
vlogan -full164 -work -sverilog -f files.sfl -kdb
vcs -full164 -top <name> -kdb
-hierarchy_info=$HIERARCHY_INFO
```

## Running Compile on Compilation Groups

Before your run the compile, make sure that same options are set on all compiler groups.

1. Configure option `set use_module_idb 1` on all compilation groups.
2. Make sure each group run maintains its own `simv.daidir`, and `ucdb` directories.
3. Use option `-compile_only 1` for each group run.
4. Specify the IDC file with all the required IICE settings and clock definitions. The IDC should only be given to the SoC group representing the top module.

```
set dirLoc "/bottom_up/GRP_N"
database load db_grp_N -autocreate -technology HAPS-100
source /bottom_up/options.tcl

launch uc -utf ${dirLoc}/uc_data/run.utf -ucdb -v 2.0

Specify IDC only for group representing "top" module
Used for specifying debug IP settings

run compile -ucdb ${dirLoc}/ucdb_grp_N -out grp_N_c0 \
 -top_module grp_N \
 -compile_only 1
 -idc <path>
```

5. Run the main link step using the `run compile` command.
6. Import all group runs using `-srcstate` option.
7. Specify the top module details using the `-top_module` command.

```
run compile -srcstate {/bottom_up/GRP_1/db_grp_1|grp_1_c0} \
 /bottom_up/GRP_N/db_grp_N|grp_n_c0} \
 -out soc_top_c0 \
 -top_module top
```

## Debugging with the Verdi Software

Before you start, specify the RTL KDB directory pointing to the `simv.daidir` under the pre-requisite run location. Make sure that RTL KDB is generated correctly for the entire design.

1. Run the prototyping and debug flow.

2. Run synthesis.
3. Export data for Runtime, hardware capture, FSDB generation and Verdi debug.

```
export runtime -path ${dirLoc}/runtime_soc_db
```
4. Generate Verdi correlation database (CRDB).

```
launch verdi -run_dir ${dirLoc}/runtime_soc_db/system/verdi -crdbgen -fast_crdbgen -fsdbtype edif
```

## Limitations

- Supports only the compilation groups run with Unified Compile. You cannot use the compilation groups run with standard compiler partition flow.
- The SRS bypass flow is not supported for RTL instrumentation changes.

## Using the Bypass Flow (Standard Compiler)

Some design cycle iterations are triggered by small changes that do not affect partitioning results. For these kinds of updates, you can save on runtime by using the RTL bypass flow. With this flow, re-partitioning is not required; you only need to rerun the single-FPGA databases that have been modified. The bypass flow ensures a faster time to bit files and the hardware, and preserves the prototype implementation.

1. Make the RTL change, and then recompile the top level of the design.
2. Determine if you can use the bypass flow.

Based on your knowledge of the design, determine which FPGAs are affected by the RTL changes. The best candidates for the bypass flow are designs where the changes are small, and localized within an FPGA. If partitioning is affected or routing must be modified, do not use this flow.

3. Use an incremental compile run to recompile each modified FPGA.
  - Recompile the RTL. If you have enabled the distributed\_compile option, this will be fast and incremental for small RTL changes. Without distributed\_compile set, it can still be faster than a full compile.

- In FPGAs that are affected by the RTL change, create a new compile state by relinking the top level compile state with the prior FPGA compile state. This will bring in the modified RTL.
- Make sure to specify the top module name.

In the following example, db0|cmod is the top-level compile state, and the top-level module name is fb1\_uB:

```
run compile -out incr1 -srcstate c0 -srcstate ../../db0|cmod -top_module fb1_uB
```

If you see errors, the bypass flow cannot be used, and you must rerun the entire flow, including partitioning. You get errors if the incremental flow is not allowed; for example if the tool changes one of the modules that was also changed in RTL.

4. Follow the normal design flow, and map, place, and route the modified individual FPGAs.

# Specifying Constraints

Constraints are optional. At an initial stage of the design, you might not use any constraints, but you can add them to refine your prototype. There are different kinds of constraints, which are specific to certain database states.

Some constraints only work for particular database states. The table below lists the constraints and the design stages where they apply:

| Constraint | Valid State                                                | Description                                    |
|------------|------------------------------------------------------------|------------------------------------------------|
| CDC        | Compile                                                    | Compiler constraints file                      |
| FDC        | Pre-map, pre-partition, system route, system generate, map | FPGA design constraints file                   |
| UPF        | Pre-map, pre-partition                                     | Power constraints file (Standard compile only) |
| IDC        | Compile, pre-instrument, pre-map, pre-partition            | Instrumentor constraints file                  |
| PCF        | Partition, system route                                    | Partition constraint file                      |

See these topics:

- [Adding Constraints to a Design Database](#), on page 167
- [Specifying Directives in a CDC File](#), on page 168

## Adding Constraints to a Design Database

This procedure describes a general method to add any kind of constraint to a design database.

1. Put the constraints in a constraints file.

Constraint file formats vary. The tool reads some file formats directly, like upf. Others are easily converted. For example, you can take standard Synopsys SDC timing constraints defined for your ASIC design and save them in an FDC file, which uses the same syntax. See [Converting Constraints](#), on page 55.

2. Add the constraints file to the design by using the appropriate argument to the run command at the appropriate database state.

For example, run map -fdc <file> specifies design constraints to be used when mapping the design. Check the command syntax for specific run command for details.

Some constraints can only be added at certain design states. The following table lists some constraints you can enter at different stages of the design:

| Command                | Constraints File                                                                                                                                                                                                                                                                                                                                                   |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| run compile            | <ul style="list-style-type: none"> <li>• cdc (Compiler directives)<br/><a href="#">Specifying Directives in a CDC File</a>, on page 168</li> <li>• idc (Instrumentation)<br/><a href="#">Instrumenting the Design for Debug</a>, on page 641</li> </ul>                                                                                                            |
| run pre_partition      | <ul style="list-style-type: none"> <li>• fdc (Synthesis design constraints)</li> <li>• upf (Low power constraints)<br/><a href="#">Including UPF Specifications</a>, on page 827</li> <li>• idc (Instrumentation)<br/><a href="#">Instrumenting the Design for Debug</a>, on page 641</li> </ul>                                                                   |
| run partition          | <ul style="list-style-type: none"> <li>• pcf (Partition constraints)<br/><a href="#">Defining Constraints in PCF Files</a>, on page 241</li> <li>• tss (Auto-partitioning constraints)<br/><a href="#">Defining the Target System in a Detailed TSS File</a>, on page 189 and <a href="#">Exploring Partition Choices with a Basic TSS</a>, on page 185</li> </ul> |
| run system_generate    | fdc (Synthesis design constraints)                                                                                                                                                                                                                                                                                                                                 |
| run system_route       | fdc (Synthesis design constraints)                                                                                                                                                                                                                                                                                                                                 |
| run pre_map<br>run map | <ul style="list-style-type: none"> <li>• fdc (Synthesis design constraints)</li> <li>• upf (Low power constraints)<br/><a href="#">Including UPF Specifications</a>, on page 827</li> <li>• idc (Instrumentation)</li> </ul>                                                                                                                                       |
| launch vivado          | par (Place-and-route constraints)                                                                                                                                                                                                                                                                                                                                  |

## Specifying Directives in a CDC File

A cdc file provides a convenient way to specify supported directives, without making changes to your HDL files. Use this procedure to create a cdc file and specify directives:

1. Create a constraints file with a `cdc` extension that contains the Tcl directives you want.

You can specify the compiler directives on views, entities, architectures and modules. See [Controlling the Compiler Run with Options and Constraints, on page 92](#) for a table that includes `cdc` directives. You can also refer to the description of a directive; if there is a `cdc` example, it is supported.

2. Use the following syntax for the directives you want. Use the syntax that matches the HDL source code you are using.

VHDL    **`define_directive {v:[libraryName.]entityName[(architectureName)]} {directive} {value}`**

Verilog    **`define_directive {v:[libraryName.]moduleName} {directive} {value}`**

The following example sets the `syn_black_box` attribute on all architectures of the sub entity in the `MyLib` library:

```
define_directive {v:MyLib.sub} {syn_black_box} {1}
```

You must specify the attribute or directive on a view (`v:`). The `libraryName` and `architectureName` arguments are optional. If you do not specify a library, the tool defaults to all design libraries. If you include an architecture, make sure to enclose it in parentheses. Note that Verilog objects are case-sensitive, but VHDL objects are not.

If you are applying directives on a net, make sure to include the pipe (|) character, as shown in the examples below:

```
define_directive {v:work.sub} syn_rename_module {sub_new}
define_directive {n:work.sub_new|temp1} {syn_keep} {1}
define_directive {n:work.test|a} {syn_direct_reset} {1}
define_directive {n:work.ones_cnt|vout_reg} {syn_allow_retiming} {1}
define_directive {n:work.test|dout_t} {syn_DSPstyle} {dsp}
define_directive {n:work.fsm|state} {syn_encoding} {one_hot}
define_directive {n:work.test|a} {syn_direct_set} {1}
define_directive {n:work.mult|mult_i} {syn_multstyle} {logic}
define_directive {n:work.pipeline|temp2} {syn_pipeline} {1}
define_directive {n:work.RAMB4_S4|mem} {syn_ramstyle} {select_ram}
define_directive {n:work.norep|DriveA} {syn_replicate} {true}
define_directive {n:work.d_p|h_data_pip_i} {syn_srlstyle} {extractff_srl}
```

3. Add the directives file when you compile the design, using the `-cdc` or `-cdclist` options to the `compile` command.

When the design is compiled, the tool passes all active cdc files to the compiler. The compiler references the object names in these files with the original RTL objects and assigns the corresponding directives.

4. Check for constraint errors:

- Enter the `report constraint_check` command to run the constraint checker. This command generates a report called `constraint_check.rpt`, which lists the constraints that were applied, not applied or ignored. It also generates a `syntax_constraint_check.rpt` file, which reports syntax error with the constraints. To run just the syntax check, use `report constraint_check -syntax_only`.
- Analyze the design and adjust constraints. It is recommended that you use more relaxed constraints for initial runs.

# Setting Options

You can set options for run commands at different points in the design cycle to influence the results of the run. Some options only affect a certain database state or run operation, while others affect multiple states.

## Methods for Setting Options

There are different ways to set options:

| Method                                                                                                            | Description                                                                                                                                                                         |
|-------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| option set commands in a Tcl file                                                                                 | A Tcl file that contains multiple options, which can be reused in multiple sessions.                                                                                                |
| option set command typed in the tool console                                                                      | Does not persist beyond the current session. Not ideal for multiple option settings.                                                                                                |
| Options Editor from the GUI (  ) | Easy way to create an options file, check current option values, edit them, or view differences from previous run.                                                                  |
| Design Intent button in the GUI<br>design_intent command                                                          | Preset options for either fast turnaround or performance. Preset options can also be customized. See <a href="#">Synthesizing Based on Design Intent</a> , on page 493 for details. |

## Working with Tcl Options Files

The best way to manage options is to set them in a Tcl file.

1. Create the Tcl options file, using one of these methods:
  - Start with a template file of options and edit it to suit your needs. The tool installation includes template files, in the directory shown below. To open one of these files and use it as a starting point, specify it with the design\_intent command:

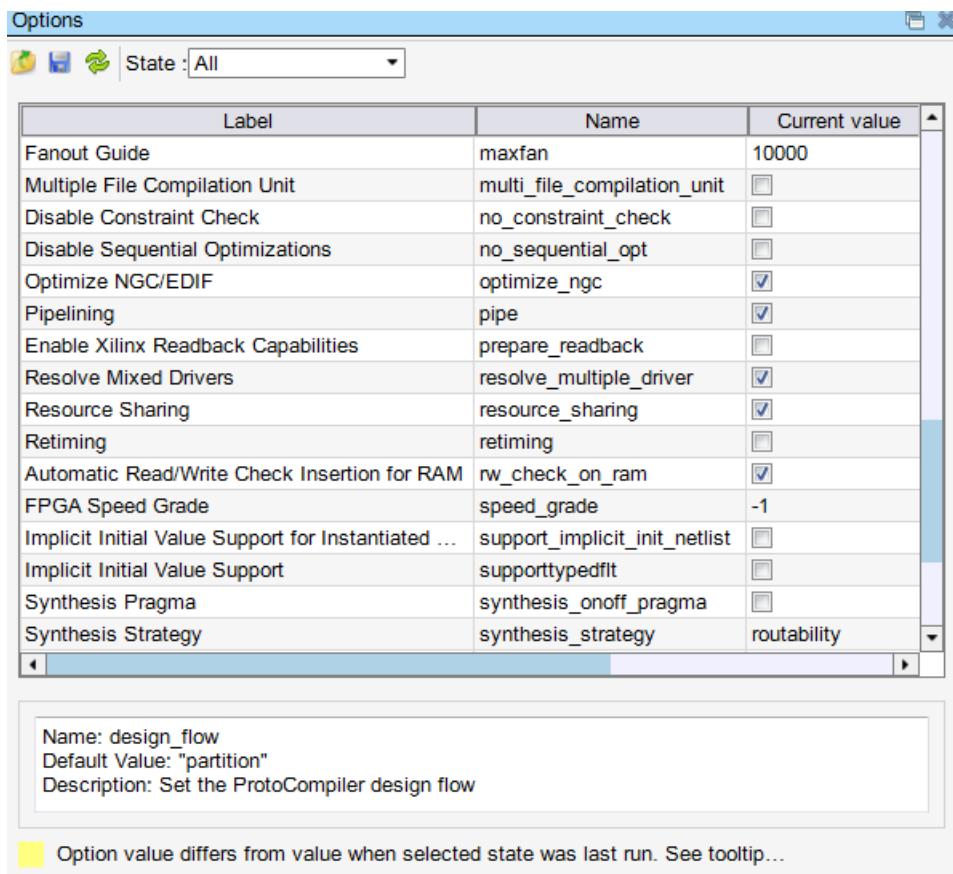
```
design_intent -custom installDir/lib/design_intent/protocompiler/timing_qor.tcl
```

If you do not specify a file with this command, the tool opens the fast\_turnaround.tcl file by default. To see a list of available Tcl options files, use the design\_intent -list command.
  - Type the options manually into a Tcl file, using this syntax for the option set command:

option set *optionName* *optionValue*

For example: option set synthesis\_strategy fast. To see the available options, enter the option list command in the Tcl window, or refer to [List of Options, on page 82](#) of the *Command Reference Manual*.

- Click the Options Editor icon in the GUI ( ). The default view shows the options for the current database state. You can set option values and then click the Save icon at the top of the tab to save a file based on the current database options. Alternatively, you can first click Show All Options at the bottom right before setting the options and saving the file.



Create a single file for all the options, or create multiple option files to be used with different database states. For example, you might set these options specifically for the map stage:

```
option set enable_prepacking 1
option set synthesis_strategy routability
option set fanout 10000
option set retiming 1
```

2. Source the options file before issuing a run command: run compile, run pre\_partition, run partition, run system\_route, run system\_generate, run pre\_map, run map.

```
source ./myCompileOptions.tcl
```

Options are not written to the database and must be sourced in order to take effect. If you are using multiple options files, make sure to source the appropriate options file to set the options. If you re-open the GUI, make sure to source the options again, as they are not stored.

To view option values, use the commands described in [Viewing Option Settings and Editing Them, on page 174](#).

3. To set options automatically on startup, do the following:

- Set the options you want in a Tcl file called .synopsys\_pc.setup.
- Save the file in one of the locations listed in the table below. The tool automatically reads options set in this file at startup. It reads the files in the order specified in the table above. After reading the file, the tool uses the specified option setting as the default.

| Setup File Location                        | Directory Description                                                                                          |
|--------------------------------------------|----------------------------------------------------------------------------------------------------------------|
| <code>installDir/.synopsys_pc.setup</code> | Installation. Sets the option or default in the install.                                                       |
| <code>~/.synopsys_pc.setup</code>          | Sets the option or default for your personal setup.                                                            |
| <code>./.synopsys_pc.setup</code>          | Sets the option or default for the working directory, which is the directory from where you launched the tool. |

4. To automatically use different options files for different experimental runs, follow these recommendations:
  - Create separate .synopsys\_pc.setup files with the different option settings for each of the databases, and save them to the different database directories.

- For the different runs, launch the tool from the locations of the `.synopsys_pc.setup` files. This ensures that the options are set automatically for that run, and no files need be sourced.

## Viewing Option Settings and Editing Them

The tool provides ways to view and edit options from the command line or from the GUI.

1. To view the value of an individual option, use the Options Editor as described in the next step, or use these commands:

| To View ...                          | Command                                                                                                                                                                           |
|--------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Current value of an option           | <ul style="list-style-type: none"><li>• <code>option get <i>optionName</i></code></li><li>• <code>option list</code></li><li>• Click the Options Editor icon in the GUI</li></ul> |
| Current default value for the option | <code>option get_default <i>optionName</i></code><br>(No return value indicates the default)                                                                                      |
| Value used to run the database state | <code>database query_state -option <i>optionName</i></code><br>either on its own, or as part of a script.<br>See <a href="#">Querying Incremental Results , on page 520</a> .     |

Note that the option value for the loaded database might differ from the current value. The two commands listed above display the current value which will be used going forward for the next run command, and the value that was used to generate the current database. For syntax details, refer to the *Command Reference*.

2. To view all the option values for a particular database state, select the state and then open the Options Editor by clicking the icon in the GUI ().

The editor shows the relevant options for that database state and their current values. If the value of an option has changed since the previous run, it is highlighted.

3. To view all the current option values, open the Options Editor and click Show All Options in the editor.

The editor shows all the options and their current values.

4. To reset a current option value to the value it had when you last ran the database state, create a script like the following example, and run it.

```
proc restore_state_options {} {
 foreach o [option list] {
 set cur [option get $o]
 if {[catch {set prev [database query_state -option $o]}]} continue
 if { "$cur" == "$prev" } continue
 puts "Changing option $o to $prev" option set $o $prev
 }
}
```



## CHAPTER 4

# Partitioning the Design

---

This chapter describes partitioning methodology and various commands to partition your design.

- [Overview of Partitioning](#), on page 178
- [Setting up Files for Partitioning](#), on page 183
- [Defining the System in a TSS File](#), on page 184
- [Generating TSS Files Using TSS Builder](#), on page 228,
- [Defining Constraints in PCF Files](#), on page 241
- [Working with HAPS Systems](#), on page 269
- [Working with HAPS-100 Modules](#), on page 296
- [Using run pre\\_partition to Define Partitions](#), on page 324
- [Partitioning the Logic](#), on page 328
- [Analyzing Partition Result Files](#), on page 358
- [Using Time Domain Multiplexing \(TDM\)](#), on page 372
- [Using Hierarchical Partitioning](#), on page 406
- [Running System Route for the Top Level](#), on page 412
- [Generating FPGAs](#), on page 416
- [Implementing Individual FPGAs](#), on page 426
- [Using Multi-Design Mode \(MDM\)](#), on page 437

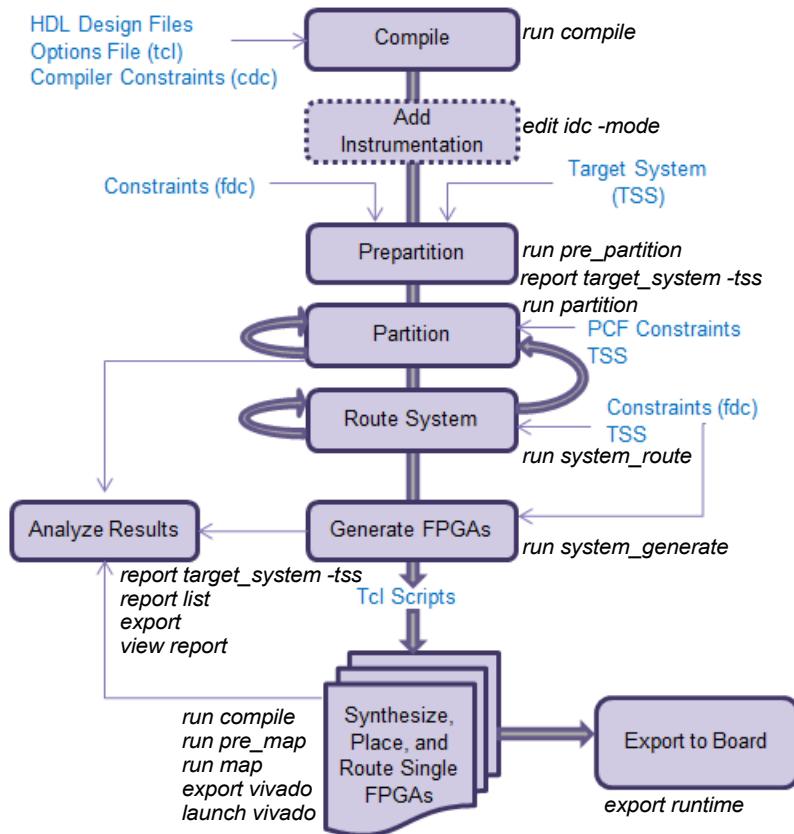
# Overview of Partitioning

Partitioning lets you start with a large design and partition it into multiple FPGAs. This is the basis of multi-FPGA design.

See these topics:

- [The Partitioning Flow](#), on page 178
- [Design Methodology for Partitioning](#), on page 179

## The Partitioning Flow



The figure shows the complete partitioning design flow and the main commands associated with each state. Partitioning itself consists of four states, each with an associated run command: pre-partition, partition, route system, and generate FPGAs. Although the diagram shows iterations at the partitioning stages only, you can re-iterate and rerun any stage in the design because of the flexible database model.

For more information about the steps shown in the preceding figure, see the table below:

|                                              |                                                                                                                                                                      |
|----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Overview of Partitioning Flow                | <a href="#">Design Methodology for Partitioning</a> , on page 179                                                                                                    |
| Compile                                      | <a href="#">Working with a Design Database</a> , on page 58<br><a href="#">Adding Design Files</a> , on page 96<br><a href="#">Compiling the Design</a> , on page 77 |
| Add Instrumentation                          | <a href="#">Instrumenting the Design for Debug</a> , on page 641                                                                                                     |
| Pre-partition                                | <a href="#">Setting up Files for Partitioning</a> , on page 183<br><a href="#">Using run pre_partition to Define Partitions</a> , on page 324                        |
| Partition                                    | <a href="#">Partitioning the Logic</a> , on page 328<br><a href="#">Using Hierarchical Partitioning</a> , on page 406                                                |
| Route System                                 | <a href="#">Running System Route for the Top Level</a> , on page 412                                                                                                 |
| Generate FPGAs                               | <a href="#">Generating FPGAs</a> , on page 416                                                                                                                       |
| Synthesize, Place and Route Individual FPGAs | <a href="#">Implementing Individual FPGAs</a> , on page 426<br><a href="#">Running the Implementation Flow</a> , on page 471                                         |
| Export to Hardware                           | <a href="#">Preparing to Run on the Hardware</a> , on page 614                                                                                                       |
| Analyze Results                              | <a href="#">Analyzing Results</a> , on page 512                                                                                                                      |

## Design Methodology for Partitioning

Partitioning logic is an iterative process, where you refine results until you get what you want. Your goal is to find an optimal assignment of cells and ports, so that the connections effectively use the available routing resources.

Typically, this task is time-consuming and difficult, because you have to balance two inter-dependent considerations that influence and affect each other: the hardware topology and the design characteristics. The hardware

configuration determines the functioning of the prototype, but design characteristics determine the best hardware configuration. The automatic partitioner (APTN) helps automate this task for faster iterations.

## Overview of Using the Automatic Partitioner

The partitioning approach described here provides a fast and flexible solution that allows for a high level of automation. The APTN flow consists of two phases: in the first, you quickly iterate through multiple experimental runs and refine results until you are satisfied; then, you implement the partitioning solution you decided was best.

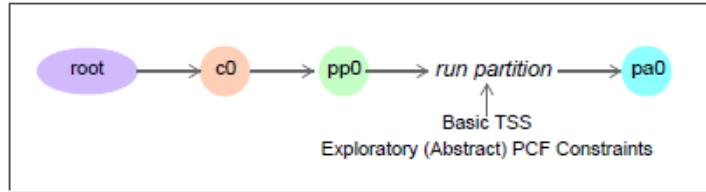
1. Set up an initial target system specification (TSS) that models the available hardware resources.
  - At a minimum, this TSS file must define the FPGAs. Additionally, you can define the clocks; the port connections and daughter boards; and inter-FPGA cabling in this file, depending on your design restrictions and your knowledge of the architecture. See *Exploring Partition Choices with a Basic TSS*, on page 185.
  - For trace names and connection information, use the report target\_system -tss command. See *Checking the Target Specification (TSS)*, on page 358 for details.
2. Refine the hardware setup, by defining exploratory constraints in pcf file.

It is recommended that you separate different kinds of constraints into multiple pcf files, so that you can use them in a modular fashion. For example, create an abstract.pcf and a setup.pcf file. See *Using Abstract PCF Commands for Partition Exploration*, on page 242 for information about creating these files.

3. Start with a pre-partitioned database state, and partition it specifying the TSS file and the pcf setup constraints. For example:

```
run partition -tss abstract.tss -pcf setup.pcf -pcf abstract.pcf
```

See *Partitioning the Logic*, on page 328 for details about generating the partition database state.



When you use a basic TSS, the partitioner generates a partitioned database state, called `pa0` by default. You cannot proceed to the next stage of design, system routing, from this database state because it was generated with a basic TSS file and exploratory constraints in the PCF file. To partition the design, you require a complete TSS file definition of the hardware, with no abstract constraints.

The partitioner also generates a log file and a report. If you have conflicting constraints in multiple files, the partitioner implements what was specified in the last file.

4. Examine the log and report (`prf`) files using the `view report` command, to determine what constraints to add or adjust for the next run.

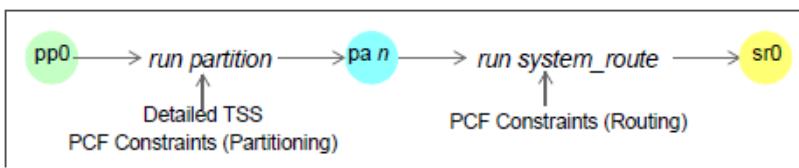
See [Methodology for Analyzing Partitioning at Different Stages, on page 360](#) and [Checking Partitioning Reports, on page 367](#) for details.

5. Rerun partitioning with updated TSS and PCF files.
  - Rerun partitioning with these files and the TSS file.
  - Analyze the log and report files.
  - Repeat this set of steps until you are satisfied with the results. Partitioning with a basic TSS and pcf constraints is quick, so you can easily loop through several iterations.
6. When you are satisfied with the results, run partitioning with a complete TSS file, and run system routing.
  - Create a detailed TSS file for system routing. Convert the pcf constraints needed to implement the partitioning solution you want (step 5) into `board_system*` commands in a new, detailed TSS file, and remove the original abstract pcf constraints you used for earlier runs. The detailed TSS file is required to proceed beyond partitioning.

See [Defining the Target System in a Detailed TSS File, on page 189](#) for more information about creating this file, and [Target System](#)

[Specification Commands, on page 243](#) in the *Command Reference Manual* for command descriptions.

- Run partitioning with the detailed TSS file. This generates a partitioned database state where you can run system routing.
- Specify any constraints to guide system-level routing in a PCF file (route.pcf, for example) and specify it when you run system routing.
- Run system routing as described in [Running System Route for the Top Level, on page 412](#).



7. To lock down a final partitioning solution, follow these steps:

- When you have a solution you want to save, use the `export file` command to export the constraints file and TSS, if you have one.
- Specify the exported constraints file and the detailed TSS when you rerun partitioning or system route.

See [Locking Down Partitions, on page 352](#) for information about using pcf files for incremental flows.

# Setting up Files for Partitioning

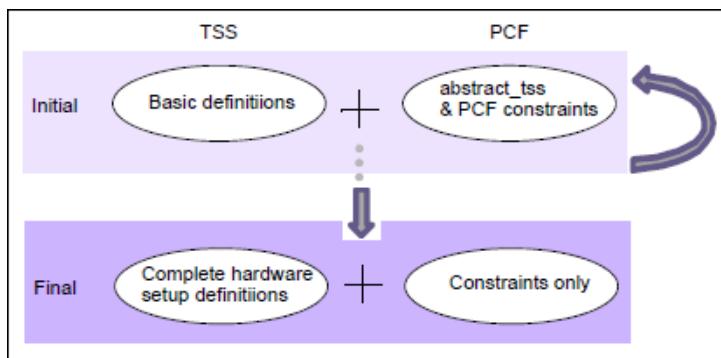
To partition a design, you must have these files:

| Files                                                                                                                                                                                                                                                 | Format/File                   |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------|
| HDL logic design files. See <a href="#">Adding Design Files</a> , on page 96 for setup information.                                                                                                                                                   | Verilog/VHDL or a combination |
| One or more partition constraint files to explore partitioning options. See <a href="#">Defining Constraints in PCF Files</a> , on page 241 for setup information.                                                                                    | .pcf                          |
| Board description or target system specification (TSS) file. This is the software model of the hardware configuration, and can be basic or detailed. See <a href="#">Defining the System in a TSS File</a> , on page 184 for details about this file. | .tcl                          |

# Defining the System in a TSS File

The target system specification (TSS) file is a Tcl file that defines the hardware configuration. This is the only definition of the hardware in the prototyping software, so it is important that the hardware be fully and accurately modeled. The hardware configuration is described using `board_system` commands that define the hardware board, interconnect, clocks, resets, and voltage regions.

At the initial exploratory design phase, the TSS tcl file contains basic partition information, and is used in conjunction with `abstract_tss` constraints entered in a PCF file. As you refine your partition choices, the TSS file specifies your final partition choice and the actual hardware. Partitions that can be routed are based on the specifications in this file.



For more information about using the TSS file for these purposes, see these topics:

- [Exploring Partition Choices with a Basic TSS](#), on page 185
- [Defining the Target System in a Detailed TSS File](#), on page 189
- [Defining Clocks in the TSS File](#), on page 194
- [Defining Hardware Interconnect](#), on page 196
- [Defining Connections with `board\_system\_create -interconnect`](#), on page 200
- [Defining Daughter Boards](#), on page 204
- [Defining Connections with Auto-Cabling](#), on page 197

- [Moving from Abstract PCF to TSS: Flow Example](#), on page 220
- [Validating TSS Cable Connections with conspeed\\_hstdm](#), on page 225
- [Validating the TSS Against the Hardware](#), on page 226

For additional information, see these topics:

- [Design Methodology for Partitioning](#), on page 179
- [Working with HAPS Systems](#), on page 269

## Exploring Partition Choices with a Basic TSS

The basic TSS file is intended to be a quick mechanism for hardware target exploration. It contains basic system configuration commands, and is used in conjunction with abstract pcf constraints to explore partitioning choices. As you iterate through partitioning, you update the TSS file to reflect your choices, until you run system routing with your final choices and a detailed TSS file. For an example that illustrates the abstract flow, see [Moving from Abstract PCF to TSS: Flow Example](#), on page 220.

The following procedure describes the basic TSS commands for system configuration. Detailed descriptions for these commands are in [Chapter 4, Target System Specification Commands](#) in the *Command Reference Manual*.

1. Create a basic TSS file:

- Use the `launch tss` command to open the TSS editor. If you use the editor to enter the TSS commands, you can take advantage of the built-in syntax help.
- Enter `board_system_create` commands to define the systems. If you type `help` in the TSS editor, it displays help and syntax for the TSS commands. For an initial run, all that is required is a definition of the correct number of FPGAs. Here are some examples:

```
board_system_create -haps -name haps-70sys
board_system_create -add HAPS70_S48 -name fb1

board_system_create -haps -name haps80sys
board_system_create -add HAPS80_S104 -name fb1
```

To get the exact keywords, use the `board_system_list -objects` command.

- Specify the speed grade with the `board_system_create -speed_grade` command in the TSS file. It is recommended that you do this for each system added.

```
board_system_create -add HAPS80_S26 -name FB1 -speed_grade {-2-e}
```

To view the valid speed grades for the current system (which is set with option `set technology`) use option `get valid_speed_grades`. Specifying a speed grade sets it for all FPGAs on the HAPS system, but you can specify different speed grades for different systems. During the partitioning process, the speed grade and technology are written to the individual FPGA partition files for synthesis. Do not change these settings at the individual FPGA level.

For single FPGA designs, you can either define the speed grade in the TSS file or use the option `set speed_grade` command.

- Save the file.

As you proceed through more iterations, add details to the TSS, like the clock and reset plan. For more information, refer to [Defining the Target System in a Detailed TSS File, on page 189](#).

2. Set `abstract_tss` commands in the `pcf` file to define configuration schemes and I/Os that you want to experiment with. This example defines an interconnect scheme:

```
abstract_tss -add_port_bin -fpga FB1.uA -name \ ABS_PORTA -width 650
abstract_tss -add_port_bin -fpga FB1.uB -name \ ABS_PORTB -width 650
```

The `pcf` constraints do not all have to be defined at the same time or in the same file, but can be added successively to different files, as you develop the prototype. See [Defining Constraints in PCF Files, on page 241](#).

3. Partition the design with `run partition`.

This run is very fast. The run does not generate a new database state, because it is run with abstract `pcf` commands and a preliminary `tss` file. The `run partition` command has options that offer additional controls for exploratory runs. Some of them are described below. See [run partition, on page 144](#) in the *Command Reference Manual* for the complete syntax.

- Set `-effort` to low, normal, or high. The setting affects the default setting for other `run partition` command arguments, as shown below.

| Argument                 | Low          | Normal       | High              |
|--------------------------|--------------|--------------|-------------------|
| -max_trials              | 150          | 1000         | 1000              |
| -solutions               | 1            | 3            | 6                 |
| -hierarchy_dissolve_pass | 1            | 2            | 2                 |
| -multi_hop_accuracy      | 0 (estimate) | 0 (estimate) | 1 (full accuracy) |

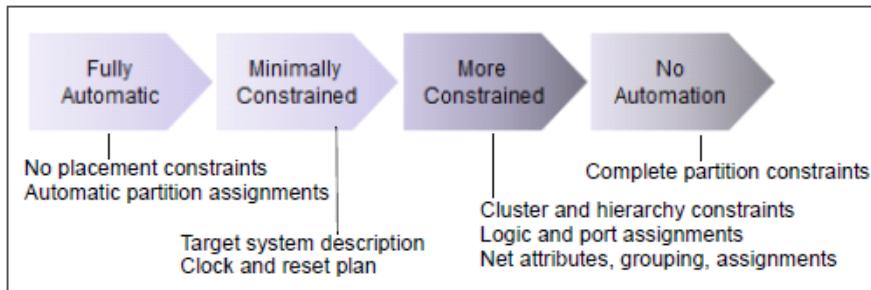
- Specify other run partition arguments to override the defaults set by -effort:

#### Argument Description

|             |                                                                                    |
|-------------|------------------------------------------------------------------------------------|
| -max trials | Specify the number of exploratory trials you want the partitioner to run.          |
| -solutions  | Specify the number of solutions to evolve in parallel during the partitioning run. |

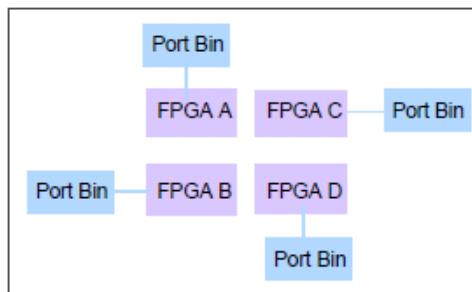
- For initial partitioning runs, you can also set the advanced options -clock\_crossing\_reduction and -illegal\_feedthrough\_reduction to none or low.

4. Adjust pcf constraints and tss definitions.
  - Change pcf constraints and rerun partitioning.
  - As you settle on design details, remove the abstract\_tss constraints from the pcf file, and add the equivalent commands in the tss file. See step 5 for typical commands added to the tss file during exploration.
  - Iterate as many times as needed to achieve the results you want. The iterative process is described in [Design Methodology for Partitioning, on page 179](#). The following figure shows how the TSS constraint definitions evolve with the prototype, becoming driven by the TSS rather than the pcf, until it is completely defined by the TSS file:



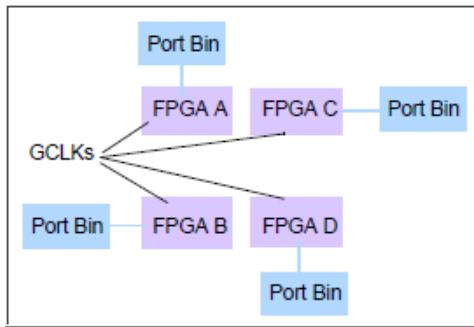
5. Define other configuration elements in the TSS file as the prototype is developed.
  - Create one bin for each FPGA. A bin is collection of target resources, a group of logical or physical elements of the HAPS system that is referenced in the TSS file. Remember that the basic TSS does not need to be set to physically realizable limits. For example, setting the number of pins required on an FPGA to a very large number gives the partitioner more freedom.
  - Create one port bin for each FPGA using the `board_system_configure -top_io` command:

```
board_system_configure -top_io {FB1.A1}
board_system_configure -top_io {FB1.B1}
board_system_configure -top_io {FB1.C1}
board_system_configure -top_io {FB1.D1}
```



- Create one or more locked bins for clocks, resets, black boxes and internal memories. You must define all bins before defining traces. Map known design clocks to HAPS pll or fpga sources, as shown below.

```
board_system_configure -clock FB1.GCLK1 pll
board_system_configure -clock FB1.GCLK5 pll
board_system_configure -clock FB1.GCLK8 fpga
```



- Define traces that connect each port bin to its corresponding FPGA.
  - Define traces that connect all bins in pairs.
  - Define traces for clocks and resets.
6. When you are satisfied with your prototype configuration, make sure that the TSS file reflects the pcf constraints you used to achieve the result, and use this detailed TSS file to partition the design and run system routing.

See [Defining the Target System in a Detailed TSS File, on page 189](#) for details about what the detailed TSS should contain. Any pcf file you have at this point must not contain any configuration constraints; it should only contain constraints for partitioning and system routing. The detailed tss file used for partitioning overrides previous tss files used for pre-partitioning, for example

For an example of the abstract flow, moving from pcf definition to tss, see [Moving from Abstract PCF to TSS: Flow Example, on page 220](#).

## Defining the Target System in a Detailed TSS File

To run system routing and create FPGA partitions, you need a detailed TSS file, which defines the actual hardware setup. If you used abstract\_tss PCF commands to explore prototype configurations ([Exploring Partition Choices with a Basic TSS, on page 185](#)), you must make sure that the final configura-

tion information from the pcf constraint file is reflected in the TSS file before you run system routing, and that the pcf file no longer contains the abstract\_tss commands used for partition exploration.

The TSS is the only definition of the hardware that the prototyping software uses as a basis for partitioning, so it is important that it be modeled completely and accurately. The detailed tss file used for partitioning overrides previous tss files used for pre-partitioning, for example.

- For a partitioned individual FPGA, the final TSS file must define the HAPS system, the global clock, the global reset, and the UMRBus connections.
- For a multi-FPGA design, the TSS must additionally define CDE chaining, and clock-left and clock-right connections from board to board. CDE is the HAPS supervisor module.[e Defining Connections with board\\_system\\_create -interconnect, on page 200](#) for more information about UMRBus chaining.

The following procedure refers to various commands; the complete syntax for each is described in [Chapter 4, Target System Specification Commands](#) in the *Command Reference Manual*. See [Example: Detailed TSS File, on page 193](#) for an example of this file.

1. Prepare to enter the TSS commands.

At the exploratory initial stage, a TSS file can be accompanied by a PCF file with abstract\_tss commands that define board configuration information that will eventually be defined in the TSS.

- For a final TSS file, make sure that there are no abstract\_tss commands defined in accompanying pcf files. If there are abstract\_tss commands defined, they take priority and you cannot complete the later partitioning phases and implement the partitions.
- Use the launch tss command to open the TSS editor. If you use the editor to enter the TSS commands, you can take advantage of the built-in syntax help. Type help to see syntax details.

2. Use the board\_system\_create command to specify the number of FPGA partitions:

- Supply a name for the target system. For example, this command defines a target system called six\_chip\_sys:

```
board_system_create -haps -name six_chip_sys
```

- Add HAPS systems and daughter boards with the -add argument. Use a separate board\_system\_create -add command to add each board.

```
board_system_create -add HAPS70_S48 -name FB1
```

For daughter boards, use board\_system\_create to define daughter board positions and connections. (See [Defining Connections with board\\_system\\_create -interconnect, on page 200](#).)

- Use the board\_system\_list -objects command to check for available systems, daughter boards, and interconnect boards:

```
board_system_list -objects
```

- To add HAPS-compliant custom daughter boards, use additional arguments to the board\_system\_create command as described in [Defining Custom Daughter Board Connections, on page 215](#).

|                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -haps_custom_db            | Specify the custom daughter board                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| -HT3   -HT2   -MGB   -HSIO | Specify the connector for the custom board: HapsTrak 3, HapsTrak II, HapsTrak MGB, or HapsTrak High Speed I/O. You must specify the list of connectors and their types.                                                                                                                                                                                                                                                                                                                                                                                                                                |
| -guts                      | <p>Create a Verilog _guts.v file that defines the connection model for the module. It must have the black box information and pin connections for the custom daughter board, and include a trace delay value (.tdv) file. See <a href="#">Example: _guts.v File, on page 218</a> for an example of the pin connections.</p> <p>Then specify the file with this argument so that traces are made available for assignment. If using a relative path, specify it relative to the pre-partition state. The contents of this file are included in the module definition for the custom daughter board.</p> |

The following example specifies a custom adapter board, with pin connections defined in the .v file defined with the -guts option:

```
board_system_create -haps_custom_db "HAPS_JUNO_ADAPTER"
-HT3 {JX1 JX2 JX3} -guts JUNO_ADAPTER.v
```

### 3. Define the board-to-board connections.

You must model the top-level I/O ports, all the hardware connections between boards, and the cabling.

- Specify board-to-board connections with `board_system_create -interconnect`, as described in [Defining Connections with `board\_system\_create -interconnect`, on page 200](#). For examples of daughter board connections, see [Examples: Daughter Board Connections, on page 208](#).
- If your hardware setup includes a HAPS-DX7 system for debug, see [Defining a HAPS-DX7 System as a Subsystem, on page 289](#) for connection details.
- If needed, use the `-disconnect` option to disconnect boards in an existing file:

```
board_system_create -disconnect fb_3
board_system_create -disconnect connC
```

4. Map known design clocks to HAPS sources with the `board_system_configure -clock` command.

See [Defining Clocks in the TSS File, on page 194](#) and [Working with HAPS Global Clocks, on page 269](#) for more information.

5. Define board system parameters.

- To view available parameters for an instance, use this command:

```
board_system_list -parameters -instance instanceName
```

- To define voltage regions, use `board_system_configure -voltage`. The following example shows typical commands for a HAPS-70 system:

```
board_system_configure -voltage {fb1.uA_V1 fb1.uA_V2 fb1.uA_V3
fb1.uA_V4 fb1.uA_V5 fb1.uA_V6 fb1.uA_V7 fb1.uA_V8
fb1.uA_V9 fb1.uA_V10 fb1.uA_V11 fb1.uA_V12} 1.2
board_system_configure -voltage {fb1.uD_V8 fb1.uD_V9 fb1.uD_V10
fb1.uD_V11 fb1.uD_V12 fb1.uD_V13 fb1.uD_V14} 0
```

- Set the system (technology) and speed grade to match the target set for the compile stage. You get a warning message if you do not define these parameter. If you did not define the technology when you created the database with `database create`, define it as an option:

```
option set technology haps-80
```

Define the speed grade in the TSS file:

```
board_system_create -add HAPS80_S26 -name FB1 -speed_grade {-2-e}
```

6. When editing an existing file, use the `-disconnect` option to disconnect extra traces or board systems:

- In the following example 400 traces are created between FPGAs uA and uB because of the `-interconnect -auto` option:

```
board_system_create -interconnect -auto -width 400
 -devices {fb_1.uA fb_1.uB}
```

If you only require 280 traces you can remove the 120 extra traces with the `-disconnect -width` option:

```
board_system_create -disconnect -width 120
 -devices {fb_1.uA fb_1.uB}
```

- Use `-disconnect` option to remove systems or interconnect boards:

```
board_system_create -disconnect fb_3
board_system_create -disconnect connC
```

## Example: Detailed TSS File

The following example of a TSS file models four devices, which are connected manually across the A13, D23, A12, and C13 connectors on a HAPS-70 S48 system, using HT3 breakout connectors and cabling. The setup also includes one PCIE interface card.

```
board_system_create -haps -name test_chip
board_system_create -add HAPS70_S48 -name fb1

#Clock Configuration Commands
board_system_configure -clock fb1.GCLK1 PLL
board_system_configure -clock fb1.GCLK2 PLL
board_system_configure -clock fb1.GCLK3 PLL
board_system_configure -clock fb1.GCLK4 fpga

#TOP I/Os
board_system_configure -top_io fb1.A1
board_system_configure -top_io fb1.B1
```

```
#FPGA Connections
board_system_create -interconnect -manual BREAKOUT_HT3
 -name conn1 -connector {fb1.A13 0 0}
board_system_create -interconnect -manual BREAKOUT_HT3
 -name conn2 -connector {fb1.D23 0 0}
board_system_create -interconnect -manual CON_CABLE_50_HT3
 -name conn3 -connector {fb1.A12 fb1.B12}
board_system_create -interconnect -manual CON_CABLE_50_HT3
 -name conn4 -connector {fb1.C13 fb1.B13}

#CDE
board_system_configure -connect {umrbus_if.CDE_OUT fb1.CDE_IN}

#PCIE interface card on fb1.uA
board_system_create -interconnect -manual RISER1_MGB
 -name mgbA1 -connector {0 0 fb1.AM1}
board_system_create -interconnect -manual PCIE4_MGB
 -name pcie_A1_j2 -connector mgbA1.J2
board_system_create -interconnect -manual PCIE4_MGB
 -name pcie_A1_j3 -connector mgbA1.J3
```

## Defining Clocks in the TSS File

The final TSS file defines clocks in terms of the hardware setup. It sets up the global clocks (GCLKs) from the HAPS global clock network and defines the sources for the global clocks. Clocks that are not mapped to HAPS clock sources can cross FPGAs and cause skew.

This is just one part of the clock definition. See [Assigning Clocks in the PCF, on page 251](#)

1. Check the clocking scheme in the hardware manual for the board system. The TSS file supports both direct and indirect clock synchronization.

Consult the hardware Reference manual for details about the clock infrastructure for the system you are using.

2. Define port bins for the FPGAs with the `board_system_configure -top_io` command.

You can use the `launch tss` command to open an interactive Tcl shell and enter the TSS commands there. If needed, use the `board_system_list` command to list available resources.

---

```
board_system_configure -top_io{FB1.A4}
board_system_configure -top_io{FB1.B1}
```

3. Specify a source (fpga, pll, or external) for GCLKs to be used for the design. For example:

```
board_system_configure -clock FB1.GCLK8 fpga
board_system_configure -clock FB1.GCLK1 pll
```

The `board_system_configure -clock` commands tie the HAPS clock sources to specific clocks on the GCLK network for the HAPS system. For step-by-step details, see [Working with HAPS Global Clocks, on page 269](#).

To complete the HAPS-aware clock definition, you must also connect the design clock nets to the GCLK network and the clock sources, using PCF constraints. For details, see [Assigning Clocks in the PCF, on page 251](#).

4. Optionally, define the clock frequency for the clock source with the `-frequency` option:

```
board_system_configure -pll FB1.PLL1_1 -frequency 10000
```

Specifying the frequency in the TSS file allows for a more complete Confpro script to be generated from the TSS file later. Alternatively, you can use `cfg_clock_set_frequency` command in Confpro to set the frequency when you configure the system.

5. Configure other connection parameters for the clocks, like feedback, the driving source for PLL, or indirect clock synchronization.

This example instantiates a CDE\_CABLE across the CLK connector pairs for a HAPS-70 series system:

```
board_system_configure -connect {fb1.CLK_RIGHT fb2.CLK_TOP}
```

The next example shows indirect clock synchronization among three HAPS-70 S48 systems:

```
board_system_create -haps -name hapsIndirectSync
board_system_create -add HAPS70_S48 -name fb1
board_system_create -add HAPS70_S48 -name fb2
board_system_create -add HAPS70_S48 -name fb3
board_system_configure -connect {fb1.CLK_RIGHT fb2.CLK_LEFT}
board_system_configure -connect {fb2.CLK_RIGHT fb3.CLK_LEFT}
board_system_configure -connect {fb1.VIRTUAL_GCLK fb2.VIRTUAL_GCLK}
```

```
board_system_configure -connect {fb2.VIRTUAL_GCLK fb3.VIRTUAL_GCLK}
board_system_configure -clock fb1.GCLK1 pll
board_system_configure -clock fb2.GCLK1 pll
board_system_configure -clock fb3.GCLK1 pll
board_system_configure -clock {fb1.VIRTUAL_GCLK1} gclk
```

## Defining Hardware Interconnect

It is very important that the TSS file model the hardware connections between FPGAs to correctly represent the physical setup. The way the interconnect is set up can greatly influence performance.

See the following:

- [Defining Connections with Auto-Cabling](#), on page 197
- [Defining Connections with board\\_system\\_create -interconnect](#), on page 200
- [Defining a HAPS-DX7 System as a Subsystem](#), on page 289
- [Examples: Daughter Board Connections](#), on page 208
- [Defining Custom Daughter Board Connections](#), on page 215
- [Validating the TSS Against the Hardware](#), on page 226
- [Working with HAPS Systems](#), on page 269

## Defining Connections with Auto-Cabling

The choice of inter-FPGA cables has an immediate effect on partitioning, and ultimately affects routability and system performance. Automatic cabling or *auto-cabling* automates the normally iterative and time-consuming process of identifying the best configuration of HT3 cables.

The alternative is to manually define the connections. See [Defining Connections with board\\_system\\_create -interconnect, on page 200](#) for details about this method.

The following procedure describes how to take advantage of auto-cabling.

1. Create an initial TSS file that contains the following basic information:

| Required Definitions           | Optional           |
|--------------------------------|--------------------|
| HAPS system/module definitions | Clock connectivity |
| Daughter board connectivity    | GCLK definitions   |
| Reserved top-level I/Os        |                    |

2. Add cable definitions based on your knowledge of the design, while keeping congestion and routability in mind.

You can define the cables in one of two ways: set abstract definitions in the PCF or TSS files, or set specific TSS definitions. With abstract definitions, you cannot specify cable length, but with specific TSS definitions, you can define the HT3 cable lengths for the connections.

- To define specific cable connections, include the `board_system_create -interconnect -manual` command in the TSS file. For example:

```
board_system_create -interconnect -manual CON_CABLE_25_HT3
 -name conn2 -connector {FB1.A23 FB2.B23}
```

When auto-cabling is enabled, the tool can modify the connection for a better result and move the cable to a different pair of connectors.

- To define specific hard connections that auto-cabling will honor, use the `-hard 1` option with the TSS `board_system_create -interconnect -manual` command. For example:

```
board_system_create -interconnect -manual CON_CABLE_25_HT3
 -name conn2 -connector {FB1.A23 FB1.B23} -hard 1
```

Auto-cabling will keep this connection as-is and not modify it.

- For abstract cable definitions, use either of these methods:

```
PCF abstract_tss -add_trace_group {FB1.uA FB1.uB FB2.uA FB2.uB} -width 200 -connect_all
```

```
TSS board_system_create -interconnect -auto -width 96 -devices {FB1.uA FB1.uB}
```

---

Do not specify a cable length with abstract definitions; auto-cabling uses 100 cm cables for all abstract connection definitions. The advantage to abstract connections is that they are easy to use.

- If there are multiple cable definitions, the order of precedence is abstract flow using PCF definitions, abstract definitions in the TSS, and lastly, defined HT3 cable connectivity.

3. Add definitions for certain cases, so auto-cabling handles them correctly.

- For HAPS-80 systems, exclude connector 12 from the scope of auto cabling, because it cannot be used for TDM. To exclude it, define it as a top I/O with the `-top_io` argument. For example:

```
board_system_configure -top_io {FB1.A12}
```

If you need to add cables on this connector, add them manually to the final TSS later, before running system route.

- Similarly, for empty locked FPGA bins that are reserved for future use, define all these connectors as `-top_io`. If a locked FPGA bin contains logic, do not exclude the connections from consideration by automatic cabling.

4. Enable auto-cabling by adding the `auto_tss_control -enable 1` command to the PCF file.

The default is to disable auto-cabling and specify all connections manually (`auto_tss_control -enable 0`), so you must specifically enable auto-cabling.

5. Partition the design.

- Partition the design using the TSS file and the PCF file.

```
run partition -pcflist pcflist.txt -out a1 -tss basicTss.tss
```

Auto-cabling runs at the partitioning stage. It keeps any cables defined with `-hard 1` as is, but changes other connections as needed to

optimize the design. It considers daughter board connections and port bins to be hard, and does not change them.

For single-FPGA designs and connections between FPGAs on the same system, auto-cabling does not alter cable lengths. For connections between systems in a multi-FPGA design, it automatically uses 100 cm cables for inter-system connections, keeping the same number of cables. Currently, you cannot specify available cable resources or the relative positions of connections.

The tool creates a new TSS file called `auto_cable.tss`, which includes the automatic cabling definitions as well as the other TSS information previously defined. If you have manual hierarchical partitions, the new TSS is generated after the second pass, after partitioning.

- Export the generated TSS file:

```
export file auto_cable.tss
```

6. Use the exported TSS file and PCF file as input to rerun partitioning.
  - Partition the design with `run partition`, using the exported TSS file and the PCF file. For example:

```
run partition -pcflist pcflist.txt -out a1 -tss auto_cable.tss
```

This example shows a typical sequence of commands:

```
database set_state {pp0} # prepartition database
run partition -pcflistpcflist.txt -out a0
export file auto_cable.tss
run partition -pcflistpcflist.txt -out a1 -tss auto_cable.tss
```

7. Check the results.

For the results, check the Automatic Cabling section of the log file after partitioning. It contains details like the number of cables between FPGA pairs, and the number of occupied and usable connectors for each FPGA.

When you are satisfied with the partitions and cable connections, continue with `system route` and `system generate`.

## Defining Connections with `board_system_create -interconnect`

After defining the component boards that make up the system, define the connections between them. It is very important that the TSS file model the hardware connections to correctly represent the physical setup. The way the interconnect is set up can greatly influence performance. There are manual and automatic methods to specify the interconnect between boards:

- |        |                                                                                                                                                                                                                                                                                             |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Manual | <ul style="list-style-type: none"><li>• Explicit interconnect specification: <code>board_system_create -interconnect -manual</code></li><li>• Implicit; defines the number of traces and leaves the connections to the tool: <code>board_system_create -interconnect -auto</code></li></ul> |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|              |                                                                             |
|--------------|-----------------------------------------------------------------------------|
| Auto cabling | <a href="#"><i>Defining Connections with Auto-Cabling , on page 197</i></a> |
|--------------|-----------------------------------------------------------------------------|

---

See the following for information about defining connections manually:

- [Specifying Connections Manually, on page 200](#)
- [Specifying Interconnections Explicitly or Implicitly, on page 202](#)
- [Defining Daughter Boards, on page 204](#)

### Specifying Connections Manually

The procedure below provides an overview of using `board_system_create -interconnect` commands to manually specify hardware board-level connections, top-level ports, and cabling. It covers both explicitly specified and implicitly specified connections.

See [\*Chapter 4, Target System Specification Commands\*](#) in the *Command Reference Manual* for the syntax for the commands mentioned. For additional hardware specifics, refer to [\*Working with HAPS Systems, on page 269\*](#).

1. Create an initial TSS file that contains the following basic information:

| Required Definitions        | Optional           |
|-----------------------------|--------------------|
| HAPS system definitions     | Clock connectivity |
| Daughter board connectivity | GCLK definitions   |
| Reserved top-level I/Os     |                    |

---

See [\*Defining the Target System in a Detailed TSS File, on page 189\*](#) for information about adding the boards to the TSS.

2. Define the interconnect between boards using explicit or implicit `board_system_create -interconnect` commands.

A *board* in this context can be a multi-FPGA HAPS system with each FPGA having a different connector designation, or daughter boards with different connectors. For details, see [Specifying Interconnections Explicitly or Implicitly, on page 202](#).

3. Define the interconnect for the daughter boards with the `board_system_create -interconnect -manual` command.

See [Defining Daughter Boards, on page 204](#) for details about connecting different kinds of daughter boards and custom boards.

4. For system-specific connections, follow these guidelines.

- If you are using a HAPS-DX7 board system for a subsystem or for validation, set up the system as described in [Defining a HAPS-DX7 System as a Subsystem, on page 289](#).
- If you are using a HAPS-80 system, see [Guidelines for HAPS-80 Interconnect, on page 285](#) for some connection guidelines.

5. When editing an existing file, use the `-disconnect` option to disconnect existing interconnections before defining new connections.

6. Define port bins and top-level I/O bins with the `board_system_configure -top_io` command:

```
board_system_configure -top_io {FB1.A1}
```

To specify multi-FPGA I/Os, first create a breakout and then specify the `board_system_configure -top_io` command on one of the breakout connectors. This example creates a trace route from TOP\_IO-A to B:

```
board_system_create -interconnect -manual BREAKOUT_HT3 -name
 conn_topa1 -connector {fb1.A1 0 0}
board_system_create -interconnect -manual CON_CABLE_100_HT3 -name
 connab -connector {conn_topa1.J1 fb1.B1}
board_system_configure -top_io {conn_topa1.J2}
```

7. Define UMRBus chaining between boards with `board_system_configure` commands, as shown below:

```
board_system_create -add HAPS70_S48 -name fb1
board_system_create -add HAPS70_S24 -name fb2
board_system_configure -connect {umrbus_if.CDE_OUT fb1.CDE_IN}
board_system_configure -connect {fb1.CDE_OUT fb2.CDE_IN}
```

When you define UMRGBus chaining, the tool automatically instantiates CDE\_CABLE as the interconnect board across connectors. If you mix different board systems, the newer systems must be at the beginning of the chain. For example, if you have HAPS-80 and HAPS-70 systems, the HAPS-80 systems must precede the HAPS-70 system or systems. Similarly, when chaining HAPS-70 and HAPS-DX systems for debug, make sure that the HAPS-DX system is at the end of the chain.

## Specifying Interconnections Explicitly or Implicitly

With `board_system_create -interconnect`, you can explicitly or implicitly define connections between boards. These methods are different from auto-cabling, where the tool picks the connections (see [Defining Connections with Auto-Cabling, on page 197](#)). A *board* in this context can be a multi-FPGA HAPS system with each FPGA having a different connector designation, or daughter boards with different connectors.

1. To directly specify the interconnect, use the `board_system_create -interconnect -manual` command.

To specify a connector, use the name of the board instance, followed by the designation for the on-board connector. For example, `fb_1.C1` indicates the C1 connector on the `fb_1` board.

The following example defines the interconnect between `fb_1` and `fb_2`, using HAPS interconnect cables. The first command defines a CON\_CABLE connection between the A2 and D1 connectors on `fb_1`. The second command defines a CON\_CABLE connection between the A4 connector on `fb_1` and the D3 connector on `fb_2`:

```
board_system_create -interconnect -manual CON_CABLE -name connB
 -connector {fb_1.A2 fb_1.D1}
board_system_create -interconnect -manual CON_CABLEX -name connC
 -connector {fb_1.A4 fb_2.D3}
```

If you get a warning, the specified target board does not support the specified connector and is physically incompatible with the interconnect board specified. Use the `board_system_list -type interconnectboard` command to see a list of available connections.

2. To use the implicit method, use the `board_system_create -interconnect -auto` command to define the number of traces required between FPGAs, and allow the tool to determine the interconnect.

You can specify multiple trace requests between devices. The software analyzes all the requirements and creates a TSS file after considering all the on-board direct traces. The following command defines 250 traces between FPGAs C and D on fb\_1:

```
board_system_create -interconnect -auto -width 250 -devices
{fb_1.uC fb_1.uD}
```

Do not use this TSS command to define more traces than are physically available at an initial stage; this will cause an error. For early-stage exploration purposes, use abstract pcf commands instead.

When you use `-auto`, do not define voltages (`board_system_configure -voltage`), or you could get an error.

3. To check the connectors and locations that the software assigns automatically, add the `report_board_system fileName` command as the last line in the `tss` file.

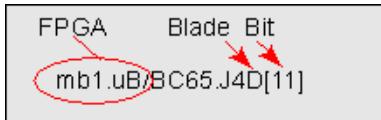
This generated file contains information about the connectors automatically selected by the tool to match your criteria. The following is a sample of the format used in this report:

| Interconnect Boards |           |                 |
|---------------------|-----------|-----------------|
| *****               |           |                 |
| Type                | Name      | Connections     |
| CON_CABLE_100_HT3   | conauto_0 | fb1.A12 fb1.B12 |

In addition, the report file generated after run `system_route`, contains detailed interconnect information:

```
@S2.2.13.1 AP414 |TraceGroup T_fb1.uB_fb1.uD_1
trace fb1_B4_D[11] -group T_fb1.uB_fb1.uD_1 -pins {
 fb1.uB/BC65.J4D[11]
 fb1.uD/BC66.J7A[9]}
```

The preceding snippet reports that Pin 11 of the fourth blade/row (D) of Connector 4 (J4) is connected to Connector 7 on FPGA D. A blade is a row of pins; the HapsTrak 3 connector has four blades, A to D.



- When editing an existing file, use the -disconnect option as needed to disconnect extra traces if needed.

In this example, there are 400 traces created between FPGAs uA and uB because of the -interconnect -auto option:

```
board_system_create -interconnect -auto -width 400
-devices {fb_1.uA fb_1.uB}
```

If you only require 280 traces you can remove the 120 extra traces with the -disconnect -width option:

```
board_system_create -disconnect -width 120 -devices {fb_1.uA
fb_1.uB}
```

## Defining Daughter Boards

This is an overview of how to add and connect Synopsys daughter boards and custom daughter boards to define the hardware setup. It includes some examples, but is not exhaustive. For details, consult the individual daughter board manuals.

- [Defining Daughter Board Connections, on page 204](#)
- [Examples: Daughter Board Connections, on page 208](#)
- [Defining Custom Daughter Board Connections, on page 215](#)

### Defining Daughter Board Connections

This procedure describes how to define connections for Synopsys daughter boards and interface cards in the tss and pcf files. For custom daughter board connections, see [Defining Custom Daughter Board Connections, on page 215](#).

For details about the syntax of commands in the following procedure, refer to the *Command Reference Manual*.

1. Add daughter boards in the tss file.

- Get a list of supported daughter boards for the HAPS system.

```
launch tss -mode shell
board_system_list -objects -type daughterboards
```

- Add daughter boards in the tss file with the `board_system_create -interconnect -manual` command.

There are some differences in how you use the command for different kinds of daughter boards. The following table summarizes how to use `board_system_create -interconnect -manual` (abbreviated to *the command*, in the table) to define different daughter boards:

|                                                                    |                                                                                                                                                                                                                                                                          |
|--------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| HAPS/HapsTrak daughter boards: SRAM, SDRAM, HTII_ADAPTER2_H3, etc. | <ul style="list-style-type: none"> <li>• Instantiate with <code>board_system_create -interconnect -manual</code>.</li> <li>• See <a href="#">Examples: Daughter Board Connections , on page 208</a>.</li> </ul>                                                          |
| ECDB (External clock distribution board)                           | <ul style="list-style-type: none"> <li>• Instantiate with the command.</li> <li>• See also</li> </ul>                                                                                                                                                                    |
| DDR3 daughter boards                                               | <ul style="list-style-type: none"> <li>• Follow guidelines in step 3 below.</li> <li>• Instantiate with the command.</li> <li>• See also <a href="#">Example 3: DDR3 Daughter Boards , on page 210</a>.</li> </ul>                                                       |
| Two-connection daughter boards, like LAB-HT3                       | <ul style="list-style-type: none"> <li>• Make sure to specify both top and bottom connectors in the tss file. If you are not using one of the connections, set it to 0.</li> <li>• See also <a href="#">Example 6: LAB-HT3 Daughter Boards , on page 212</a>.</li> </ul> |
| HAPS-compliant custom daughter boards                              | <ul style="list-style-type: none"> <li>• Use specific <code>board_system_create</code> command options, as described in <a href="#">Defining Custom Daughter Board Connections , on page 215</a>.</li> </ul>                                                             |

|                                                  |                                                                                                                                                                                                                                                                                                                                                                 |
|--------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| MGB interface cards                              | <ul style="list-style-type: none"><li>Specify two <code>board_system_create -interconnect -manual</code> commands as described in step 4.</li><li>See also <a href="#">Example 7: MGB Interface Cards with HAPS-80/HAPS-70 Systems , on page 212.</a></li><li>See also <a href="#">Using MGB Interface Cards with HAPS-100 Modules , on page 215.</a></li></ul> |
| QSFP (Quad Small-Form Pluggable) interface cards | <ul style="list-style-type: none"><li>Specify two <code>board_system_create -interconnect -manual</code> commands as described in step 4.</li><li>See also <a href="#">Example 8: QSFP+ MGB Interface Cards , on page 213.</a></li></ul>                                                                                                                        |
| Two PCI Four-Lane Cards                          | <ul style="list-style-type: none"><li>Use a separate riser card for each PCI interface card or share one riser card.</li><li>See also <a href="#">Example 9: Two PCI Express Four-Lane MGB Interface Cards , on page 214.</a></li></ul>                                                                                                                         |

- When editing an existing file, use the `-disconnect` option to disconnect existing interconnections.
- To connect DDR daughter boards, use the `board_system_create -interconnect -manual` command, making sure to follow the additional guidelines below.
  - Connect the daughter board to three HT3 connectors that lie in the same SLR (super logic region) on the HAPS system. Refer to the HAPS system reference manuals for special or limited connectors.

For example, when connecting DDR daughter boards on HAPS-70 systems, do not use connectors 16, 17, 19, or 20, because they have unconnected clock pins and different voltage limits.

On HAPS-80 systems, do not use the A1-A3 connectors, which are reserved for the DDR3\_SODIMM2R\_HT3 daughter board that is used for DTD memory storage. It is highly recommended that you do not move or remove this DDR3 memory. However, if you are not using debug and need the connectors, you can free up the A1-A3 connectors using the `board_system_create -disconnect` command.

- Configure the voltage regions by setting the VCCO of the selected HT3 connectors to 1.5 V.

DDR3\_SODIMM\_HT3 and DDR3\_SODIMM2R\_HT3 daughter boards are powered by the HapsTrak 3 connector. Daughter boards contain an ID PROM on each HapsTrak 3 connector. The ID PROM is used for

board identification and over-voltage protection, and ensures that the applied VCCO is compliant with the daughter board. Signals are impedance-controlled (50 Ohms).

- Add pcf constraints to assign DDR3 interface ports to the corresponding traces, and use this file as input when running partition and system route. Refer to the example in [Example 3: DDR3 Daughter Boards, on page 210](#) for examples of pcf constraints and tss specifications.
- 4. Instantiate MGB interface cards with two `board_system_create -interconnect -manual` commands.

The TSS is interpreted sequentially. To add MGB cards, use a two-step process with two `board_system_create -interconnect -manual` commands. The first command adds a riser card to the interface card, and the second adds the interface card to the riser. For comprehensive connector details, refer to the documentation for the interface cards.

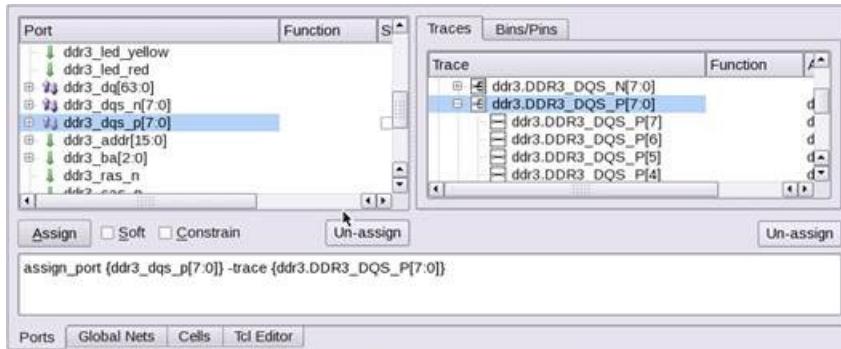
- For HAPS-100 modules, an adapter is required. See [Using MGB Interface Cards with HAPS-100 Modules, on page 215](#).
  - For MGB interface cards with HAPS-70/HAPS-80 systems, the riser card RISER1\_MGB has three connectors which are designated in order: J3, J2 and J1. J1 is connected to an available MGB connector on the system. For example, for each HAPS-80 FPGA: AM1 or AM2, BM1 or BM2 etc. J2 and J3 are connected to the other MGB interface cards. See [Example 7: MGB Interface Cards with HAPS-80/HAPS-70 Systems, on page 212](#)
  - For QSFP interface cards, use these guidelines. The eight-lane RISER8\_MGB riser card has two connectors, J2 and J1. Connect J1 to one of the MGB connectors available. The QSFPPLUS\_MGB interface card plugs into the J2 connector on the riser card. See [Example 8: QSFP+ MGB Interface Cards, on page 213](#).
5. To connect boards with HapsTrak II connections, add an adapter board and follow this example: [Example 1: HTII\\_ADAPTER2\\_HT3, on page 209](#).
  6. Assign daughter board interface ports to trace names.
    - View trace names for the daughter boards. Run pre-partition (or pre-partition and partition) with the tss file you created.
- Use the report `target_system -tss` and view report commands as described in [Checking the Target Specification \(TSS\), on page 358](#) to generate

reportpcf\_tss\_report.rpt and view the trace names for the daughter boards, respectively.

- Once you have the trace names, use `assign_port` command in the pcf file to assign interface ports from the daughter board to corresponding trace names. This is an example of trace assignment for daughter boards:

```
assign_port {ddr_odt[0]} -trace {DDR3_DDR3_ODT[0]}
assign_port {ddr_cs_n[0]} -trace {DDR3_DDR3_S_B[0]}
assign_port {ddr_we_n} -trace {DDR3_DDR3_WE_B}
assign_port {ddr_reset_n} -trace {DDR3_DDR3_RESET_B}
```

Alternatively, make trace assignments through the GUI, using the PCF editor:



The tool converts these assignments into I/O standard and pin location constraints. The pcf constraints are read and used during partition and system route.

7. Use the pcf file with the trace assignments and the tss file with the hardware setup description as input to run partition and system route.

## Examples: Daughter Board Connections

The following examples illustrate how to use the `board_system_interconnect` command to connect daughter boards. Refer to the appropriate hardware manuals for complete details.

### Example 1: HTII\_ADAPTER2\_HT3

To use HapsTrak II daughter boards with HapsTrak 3 systems an adapter board must be used. This example illustrates how to instantiate the HTII\_ADAPTER2\_H3 board.

1. In the tss file, add an adapter board and make the HapsTrak II side the top I/O, as shown here:

```
Instantiate HTII_ADAPTER2_HT3 on HAPS system
board_system_create -interconnect -manual HTII_ADAPTER2_HT3 -name
 ht2_adpt_ht3 -connector {0 FB1.A23 FB1.A22 FB1.A21}
Make the HT2 side the top I/O
board_system_configure -top_io {ht2_adpt_ht3.J1 }
```

2. Consult the documentation for the adapter board to find out how to map the HapsTrak II pins to the HapsTrak 3 pins.
3. Use report\_target\_system to find available HapsTrak 3 traces. The generated report shows information like this example:

```
#MY_FLASH connected to J1 of HTII_ADAPTER2_HT3
(which connects to HAPS on JX1, JX2, JX3)
#A[26] #MY_FLASH JX1_A[27] HTII_ADAPTER2_HT3 JX1_B[9]
#A[25] #MY_FLASH JX1_A[26] HTII_ADAPTER2_HT3 JX1_B[2]
#A[24] #MY_FLASH JX1_A[25] HTII_ADAPTER2_HT3 JX1_B[3]
#A[23] #MY_FLASH JX1_A[24] HTII_ADAPTER2_HT3 JX1_B[4]
#A[22] #MY_FLASH JX1_A[23] HTII_ADAPTER2_HT3 JX1_B[5]
```



```
#MY_FLASH connected to J1 of HTII_ADAPTER2_HT3
(which connects to HAPS on JX1, JX2, JX3)
#A[26] #MY_FLASH JX1_A[27] HTII_ADAPTER2_HT3 JX1_B[9]
#A[25] #MY_FLASH JX1_A[26] HTII_ADAPTER2_HT3 JX1_B[2]
#A[24] #MY_FLASH JX1_A[25] HTII_ADAPTER2_HT3 JX1_B[3]
#A[23] #MY_FLASH JX1_A[24] HTII_ADAPTER2_HT3 JX1_B[4]
#A[22] #MY_FLASH JX1_A[23] HTII_ADAPTER2_HT3 JX1_B[5]
```



4. Assign the pins to the correct HapsTrak 3 traces in the pcf file with the assign\_port command. For example:

```
assign_port {async_A[26]} -trace {FB1_A1_B[9]}\nassign_port {async_A[25]} -trace {FB1_A1_A[2]}\nassign_port {async_A[24]} -trace {FB1_A1_A[3]}\nassign_port {async_A[23]} -trace {FB1_A1_A[4]}
```

## Example 2: HAPS SRAM Daughter Board Connections

If you have a HapsTrak II daughter board that you would like to use on HapsTrak 3 connectors, first instantiate the adapter board as shown in [Example 1: HTII\\_ADAPTER2\\_HT3, on page 209](#), and then instantiate the SRAM. The following command adds two SRAM\_1x1 HAPS SRAM daughter boards:

```
board_system_create -interconnect -manual SRAM_1x1 -name sram1\n -connector {fb_1.A1}
```

The next example stacks the SRAM boards by specifying the connector and including the -connect\_at\_top option.

```
board_system_create -interconnect -manual SRAM_1x1 -name sram2\n -connector {sram1.J1} -connect_at_top
```

## Example 3: DDR3 Daughter Boards

The DDR3\_SODIMMHT3 and DDR3\_SODIMM2RHT3 daughter boards contain the DDR3 SODIMM memory module. If you are connecting these daughter boards, follow the connection guidelines described in [Defining Connections with board\\_system\\_create -interconnect, on page 200](#), step 4.

This is an example of the tss commands to specify a DDR3 daughter board on a HAPS-70 system:

```
Define board system\nboard_system_create -haps -name ddrcard_board\nboard_system_create -add HAPS70_S24 -name fb1\n\n# Define top-level I/Os\nboard_system_configure -top_io {fb1.A1 fb1.A7 fb1.A8 fb1.A9 fb1.A10}\nboard_system_configure -top_io {fb1.B1 fb1.B7 fb1.B8 fb1.B9 fb1.B10}
```

```

Instantiate daughter board. Connect to A4, A5, A6 on the HAPS
system, which represent HT3 connectors 3, 2, and 1 on FPGA A of
the HAPS system. The I/O standard is automatically set to SSTL.
#
board_system_create -interconnect -manual CON_CABLE_100_HT3
 -name connab -connector {fb1.A2 fb1.B2}
board_system_create -interconnect -manual DDR3_SODIMM_HT3
 -name DDR3 -connector {fb1.A4 fb1.A5 fb1.A6}

Configure voltage to 1.5V. The I/O standard is automatically set
to SSTL.
board_system_configure -voltage fb1.uA_V4 1.5
board_system_configure -voltage fb1.uA_V5 1.5
board_system_configure -voltage fb1.uA_V6 1.5

Configure clock; reset configured automatically
board_system_configure -clock fb1.GCLK0 100Mhz
board_system_configure -clock fb1.GCLK1 pll
board_system_configure -clock fb1.GCLK2 pll
board_system_configure -clock fb1.GCLK3 pll
board_system_configure -clock fb1.GCLK4 pll

board_system_configure -reset {fb1.uA_reset fb1.uB_reset} 0

```

Use `assign_port` constraints in the `.pcf` file to make corresponding trace assignments:

```

assign_port {ddr_odt[0]} -trace {DDR3.DDR3_ODT[0]}
assign_port {ddr_cs_n[0]} -trace {DDR3.DDR3_S_B[0]}
assign_port {ddr_we_n} -trace {DDR3.DDR3_WE_B}
assign_port {ddr_reset_n} -trace {DDR3.DDR3_RESET_B}
assign_port {ddr_ras_n} -trace {DDR3.DDR3_RAS_B}
assign_port {ddr_dqs_p[7:0]} -trace {DDR3.DDR3_DQS_P[7:0]}
assign_port {ddr_dqs_n[7:0]} -trace {DDR3.DDR3_DQS_N[7:0]}
assign_port {ddr_dq[63:0]} -trace {DDR3.DDR3_DQ[63:0]}
assign_port {ddr_dm[7:0]} -trace {DDR3.DDR3_DM[7:0]}
assign_port {ddr_ck_p[0]} -trace {DDR3.DDR3_CK_P[0]}
assign_port {ddr_ck_n[0]} -trace {DDR3.DDR3_CK_N[0]}
assign_port {ddr_cke[0]} -trace {DDR3.DDR3_CKE[0]}
assign_port {ddr_cas_n} -trace {DDR3.DDR3_CAS_B}
assign_port {ddr_ba[2:0]} -trace {DDR3.DDR3_BA[2:0]}
assign_port {ddr_addr[15:0]} -trace {DDR3.DDR3_A[15:0]}

assign_port {refclk_to_ddr3} -trace {DDR3.REF_CLK}

```

### Example 4: GPIO-HT3

For this daughter board, you cannot specify discrete GPIO signals as top I/Os.

```
board_system_create -interconnect -manual GPIO_HT3 -name gpio_ht3
-connector {FB1.A23}
```

### Example 5: HAPS-ECDB Clock Hub

Even if you do not use all of the available HAPS-ECDB connectors, you must initially identify each connector in the TSS file entry by specifying a value of 0 as a placeholder as shown in the example.

You cannot chain HAPS-ECDB units, but you can specify a single clock-distribution board. For a more complete example, refer to the *HAPS External Clock Distribution Board Reference Manual* on HAPS SupportNet.

```
board_system_create -interconnect -manual ECDB -name ecdb_mod
-connector {0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 }

board_system_configure -connect
{ecdb_mod.<ecdb_source_connector>.<FPGA>.<clk_direction>}
```

### Example 6: LAB-HT3 Daughter Boards

Some daughter boards have top and bottom connectors. In this case, even if you do not use one of the connections, you must define them both in the tss file. Set the unused connector to a value of 0. For example:

```
board_system_create -interconnect -manual LAB廖HT3 -name lab_ht3-connector
{FB1.A3 0}
```

### Example 7: MGB Interface Cards with HAPS-80/HAPS-70 Systems

The syntax is shown below with zero values for J3 and J2 in the first statement because no other interface cards are yet connected.

```
board_system_create -interconnect -manual riserCard -name
 riserCardConnectorName -connector {0 0 boardName.MGBConnector}
board_system_create -interconnect -manual boardName -name
 connectorName -connector {riserCardConnectorName.riserCardConnector}
```

This is an example connecting two interface cards to a HAPS-80 system with a RISER1\_MGB card.

```

Add the RISER1_MGB card to the MGB interface card. Zero values for
J3 and J2 indicate that other interface cards have not been connected
board_system_create -interconnect -manual RISER1_MGB -name conn1
 -connector {0 0 fb1.AM1}
Add the MGB interface card to the riser.
board_system_create -interconnect -manual PCIE4_MGB -name conn2
 -connector {conn1.J2}
board_system_create -interconnect -manual SATA4_MGB -name conn3
 -connector {conn2.J3}

```

See [Using MGB Interface Cards with HAPS-100 Modules](#), on page 215.

### Example 8: QSFP+ MGB Interface Cards

As with other MGB interface cards, you must first add the riser card to the FPGA and then add the MGB interface card to the riser. For QSFP (Quad Small-Form Pluggable) interface cards, the right-angled eight-lane RISER8\_MGB riser card has two connectors: J2 and J1. J1 is connected to an MGB connector on the FPGA as with other MGB interface cards, and the QSFPPLUS\_MGB interface card plugs into the J2 connector.

The first line of the example below specifies that the J1 connector of the RISER8\_MGB riser card (conn\_MGB1\_RISER8) connects to FPGA A on board fb1 through MGB slot AM1. J2 is not yet connected to anything, so its value is zero. The second line specifies that the QSFPPLUS\_MGB interface card (conn\_MGB1\_QSFP) is plugged into the J2 connector of the RISER8\_MGB riser card (conn\_MGB1\_RISER8).

```

board_system_create -interconnect -manual RISER8_MGB
 -name conn_MGB1_RISER8 -connector {0 fb1.AM1}
board_system_create -interconnect -manual QSFPPLUS_MGB
 -name conn_MGB1_QSFP -connector {conn_MGB1_RISER8.J2}

```

The following example shows two QSFPPLUS\_MGB interface card plugged into FPGA B MGB connectors BM1 and BM2:

```

Connect MGB cards for QSFP_module0
board_system_create -interconnect -manual RISER8_MGB -name connBM1
 -connector {0 fb1.BM1}
board_system_create -interconnect -manual QSFPPLUS_MGB
 -name connBM1_MGB1_J2 -connector {connBM1.J2}

```

---

```
Connect MGB cards for QSFP_module1
board_system_create -interconnect -manual RISER8_MGB
 -name connBM2 -connector {0 fb1.BM2}
board_system_create -interconnect -manual QSFPPLUS_MGB
 -name connBM2_MGB2_J2 -connector {connBM2.J2}
```

### Example 9: Two PCI Express Four-Lane MGB Interface Cards

Adding a PCI Express four-lane card requires that you first add a RISER1\_MGB card. There are two ways to configure two PCIe and riser cards, depending on whether you are sharing one riser card or using two cards:

- Using two riser cards, one for each PCIe four-lane card.

```
Add riser card 1 and connect J1 (edge connector in -connector position 3) to
an available connector on the FPGA: AM1 or AM2, BM1 or BM2 etc. The first
two -connector positions for J3 and J2 are zeros (unconnected) because
they are placeholders for daughter cards that have not yet been defined.
board_system_create -interconnect -manual RISER1_MGB -name risercard1
 -connector {0 0 fb1.AM1}
Connect the first PCI card through riser card 1
board_system_create -interconnect -manual PCIE4_MGB -name conn2
 -connector {risercard1.J2}
Add riser card 2 and connect it as described for riser card 1
board_system_create -interconnect -manual RISER1_MGB -name risercard2
 -connector {0 0 fb1.AM2}
Add PCI card 2 through riser card 2
board_system_create -interconnect -manual PCIE4_MGB -name conn3
 -connector {risercard2.J2}
```

- Using one shared riser card for both PCIe four-lane cards:

```
Add riser card 1 and connect it to the FPGA, as described above
board_system_create -interconnect -manual RISER1_MGB -name risercard1
 -connector {0 0 fb1.AM1}
Add PCI card 1 through riser card 1
board_system_create -interconnect -manual PCIE4_MGB -name conn2
 -connector {risercard1.J3}
Add PCI card 2 through riser card 1
board_system_create -interconnect -manual PCIE4_MGB -name conn3
 -connector {risercard1.J2}
```

You can only use this method if the PCIe endpoint and PCIe controller in your design and the FPGA technology support this sharing.

## Using MGB Interface Cards with HAPS-100 Modules

To use MGB interface cards with HAPS-100, a CON\_CABLE\_XX\_MGB2 and an MGB\_ADAPTER\_MGB2 are required. This is an example connecting a PCIE4\_MGB interface card to a HAPS-100 using an adapter and a 20 cm cable.

```
#MGB2 Sideband signals
board_system_configure -gpio {FB1.AM222}

#MGB2 Adapter with PCIE4_MGB
board_system_create -interconnect -manual CON_CABLE_20_MGB2 -name
mgb_cable1 -connector {FB1.AM222 NIL}

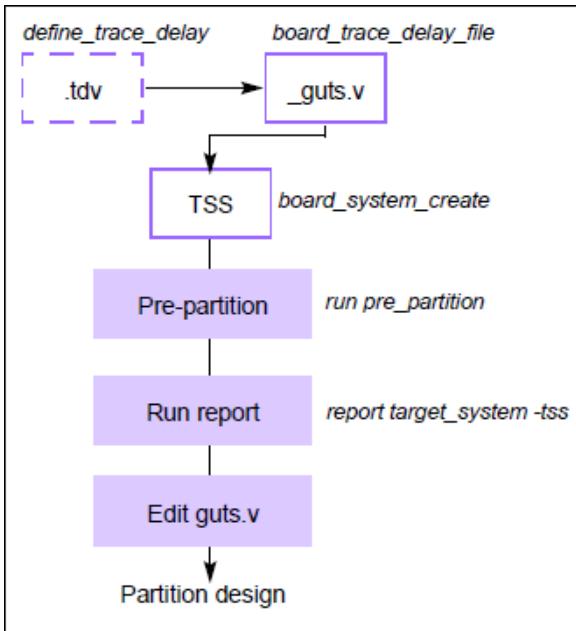
board_system_create -interconnect -manual MGB_ADAPTER_MGB2 -name
adapter -connector {NIL NIL mgb_cable1.J2}

board_system_create -interconnect -manual PCIE4_MGB -name PCIE0
-connector {adapter.J3}
```

For information about the procedure to configure the MGB\_ADAPTER\_MGB2 with PCIe sideband signals see the *HAPS® MGB\_ADAPTER\_MGB2 Sideband I/O Signals Application Note*. See also *MGB\_ADAPTER\_MGB2 Reference Manual* and the *HAPS-100 4F Reference Manual* located on HAPS SupportNet.

## Defining Custom Daughter Board Connections

Use the following procedure to define connections for HAPS-compliant custom daughter boards. For Synopsys daughter boards, see [Defining Daughter Board Connections, on page 204](#).



1. Add custom daughter boards in the tss file using the `board_system_create` command and the arguments described below.
  - Specify the custom daughter board with the `-haps_custom_db` as shown below. This command adds the initial module and port declaration, as well as the trailing `endmodule` around the `_guts.v` file, making it a complete Verilog board description.

```
board_system_create -haps_custom_db CUSTM_DB_HT3 -HT3 {JX1}
-guts ../../custom_db_ht3_guts.v
```

---

|                            |                                                                                                                                                                                                                                      |
|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -haps_custom_db            | Specify the name for the custom daughter board. Do not use connector names, like FB1.AM1, as the daughter board name. Avoid using a hierarchical names.                                                                              |
| -HT3   -HT2   -MGB   -HSIO | Specify the connector for the custom board: HapsTrak 3, HapsTrak II, HapsTrak MGB, or HapsTrak High Speed I/O. You must specify the list of connectors and their types.                                                              |
| -guts                      | Specify the file created in step 1 with this argument. If using a relative file path, specify it relative to the pre-partition state. The contents of this file are included in the module definition for the custom daughter board. |

---

You can specify multiple connectors. The following example defines two HT3 connectors (J1 JX1), one MGB connector (J2) and one HSIO connector (JH1):

```
board_system_create -haps_custom_db MY_DB -HT3 {J1 JX1} -MGB
{J2} -HSIO{JH1} -guts ../../my_guts.v
```

Define the connections for the daughter board with the `-interconnect` option. For example:

```
board_system_create -interconnect -manual CUSTOM_DB_HT3
-name CUST -connector {FB1.A5}
```

2. Create a Verilog file (`_guts.v`) that defines the connections for the daughter board.

The `_guts.v` file contains the connection model for the daughter board and must define a black box and pin connections. See [Example: `\_guts.v` File, on page 218](#) for an example. This file is used together with the `board_system_create -haps_custom_db` command (described in the previous step) to provide a complete board definition.

3. Optionally, create a file that defines trace delays for the board module and specify it with the `board_trace_delay` file attribute:
  - Create the Tcl `.tdv` trace delay file when timing accuracy is important. This file defines trace delays for the custom daughter board, as shown in the following example:

```
start_board_delay CUSTOM_DB_HT3
define_trace_delay -terminal {t:COMP.TCK} -delay 5
define_trace_delay -terminal {t:COMP.TDO} -delay 5
define_trace_delay -terminal {t:COMP.DATABUS[0]} -delay 5
define_trace_delay -terminal {t:COMP.DATABUS[1]} -delay 5
define_trace_delay -terminal {t:COMP.DATABUS[2]} -delay 5
end_board_delay
```

Use `start_board_delay` and `end_board_delay` commands to enclose the trace delay information for a daughter board.

Specify the delays for the custom daughter board traces using `define_trace_delay` commands. Use the `-terminal` argument and a prefix to identify one of the trace terminal pairs. Use the `b:` prefix for terminal ports (trace terminals inside the board definition) or `t:` for terminal pins (instances connected to the outside world). Specify the delay (`-delay` in nanoseconds).

- In the `guts.v` file, include a pointer to the `.tdv` file from the `CUSTOM_DB_HT3` module, using the `board_trace_delay` file attribute:

```
/*synthesis board_trace_delay_file = "path" */
```

Include the quotes and the `=` sign as shown, and use an absolute path.

4. Rename board traces or alias them to define actual pin locations, using the `syn_trace_attr` attribute in the `guts.v` file:
  - Run pre-partition.
  - Use the `report target_system -tss` command to list the new daughter board bins and traces. This is needed to define the pin locations. See [Checking the Target Specification \(TSS\), on page 358](#) for more information about using this report.
  - Edit the `_guts.v` file and rename or alias trace names, using the `syn_trace_attr` attribute.
5. Continue with the partitioning process (run `partition` and so on) as usual.

### Example: `_guts.v` File

This is an example of a Verilog file that defines the connections for a custom daughter board. Note that some information is added by the TSS `board_system_create -haps_custom_db` command to combine with the information in this

file and create a complete Verilog module definition. This information is shown as comments, and includes the initial module definition and the trailing `endmodule`.

```
///////////////////////////////
// Port declaration added by TSS board_system_create
// -haps_custom_db definition
//
// module CUSTOM_DB_HT3 (
// inout [13:0] JX1_A,
// inout [13:0] JX1_B,
// inout [13:0] JX1_C,
// inout [13:0] JX1_D,
// inout JX1_VRN,
// inout JX1_VRP
//);
///////////////////////////////

// Attributes for the main module, CUSTOM_DB_HT3
/* synthesis syn_partition = "board" */
/* synthesis board_trace_delay_file = "../CUSTOM_DB_HT3.tdv" */
/* synthesis syn_noprune = 1 */

// Black box signal wires and connections to HT3 ports, defined
// inside main module
wire TCK_wire = JX1_A[0];
wire TDO_wire = JX1_B[9];
wire[2:0] DATABUS_wire = {JX1_B[3],JX1_B[2],JX1_D[10]};

// Interface black box instance
CUSTOM_DB_HT3_BBOX COMP (
 .TCK (TCK_wire),
 .TDO (TDO_wire),
 .DATABUS (DATABUS_wire)
);

// Trace attributes

syn_trace_attr #(1,3,"syn_noprune=1
haps_allowed_iostandard=LVC MOS_18") TCK (TCK_wire);

syn_trace_attr #(1,3,"syn_noprune=1
haps_allowed_iostandard=LVC MOS_18") TDO (TDO_wire);

syn_trace_attr #(3,3,"syn_noprune=1
haps_allowed_iostandard=LVC MOS_18") DATABUS (DATABUS_wire);
```

```
endmodule
// For CUSTOM_DB_HT3 custom daughter board

// Definition of black box module(s)
module CUSTOM_DB_HT3_BBOX (
 inout TCK,
 inout TDO,
 inout [2:0] DATABUS
);

// Attributes for black box module
/* synthesis syn_black_box */
/* synthesis syn_partition = "black_box=CUSTOM_DB_HT3_BBOX" */
/* synthesis syn_noprune = 1 */

///////////////////////////////
// Trailing endmodule added by TSS board_system create definition
/////////////////////////////
```

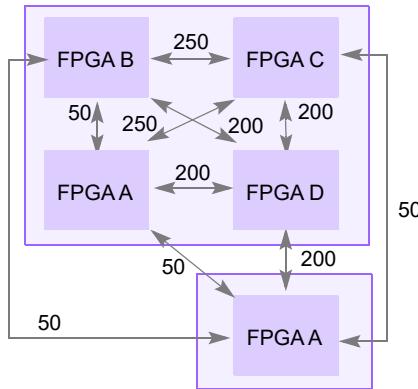
## Moving from Abstract PCF to TSS: Flow Example

The example shown below illustrates the abstract flow process, as you move a design from initial exploratory trace group descriptions in the PCF file to the final hardware interconnect description in the TSS file.

- [Step 1: Define Initial Interconnect in PCF, on page 220](#)
- [Step 2: Check Results After Running with Initial Interconnect Definition, on page 221](#)
- [Step 3: Edit Interconnect Definition and Rerun Exploratory Partitioning, on page 223](#)
- [Step 4: Transfer Final Results to TSS and Run Partitioning, on page 223](#)

### Step 1: Define Initial Interconnect in PCF

This is the initial example design:



- Define the corresponding FPGA-to-FPGA channels in the pcf file:

```
abstract_tss -add_trace_group {fb_1.uA fb_1.uB} -width 50
abstract_tss -add_trace_group {fb_1.uA fb_1.uC} -width 250
abstract_tss -add_trace_group {fb_1.uA fb_1.uD} -width 200
abstract_tss -add_trace_group {fb_1.uA fb_2.uA} -width 50

abstract_tss -add_trace_group {fb_1.uB fb_1.uC} -width 250
abstract_tss -add_trace_group {fb_1.uB fb_1.uD} -width 200
abstract_tss -add_trace_group {fb_1.uB fb_2.uA} -width 50

abstract_tss -add_trace_group {fb_1.uC fb_1.uD} -width 200
abstract_tss -add_trace_group {fb_1.uC fb_2.uA} -width 50

abstract_tss -add_trace_group {fb_1.uD fb_2.uA} -width 200
```

- Run partitioning.

## Step 2: Check Results After Running with Initial Interconnect Definition

The Bin Usage Summary shows acceptable area utilization for individual FPGAs:

**QS4.1 AP140 | Bin Usage Summary**

| FPGA_Bin | Cells | Lock | LUT    | % DFF | % BRAM | % DSP | %   |
|----------|-------|------|--------|-------|--------|-------|-----|
| mb_1.uD  | 1101  |      | 855770 | 69%   | 274421 | 11%   | 579 |
| mb_1.uC  | 8     |      | 791127 | 64%   | 206801 | 8%    | 163 |
| mb_1.uB  | 29    |      | 710082 | 57%   | 173913 | 7%    | 274 |
| mb_1.uA  | 16    |      | 970890 | 78%   | 300155 | 12%   | 466 |
| mb_2.uA  | 214   |      | 587959 | 47%   | 318225 | 13%   | 628 |

However, the Global Route Summary indicates that the maximum TDM ratio is 16:

```
QS5.4 AP267 | Global Route Summary
DN:AP367 : | Routed 15049 Nets
DN:AP268 : | Maximum TDM Ratio: 16
DN:AP368 : | Feedthroughs: 0
DN:AP270 : | Unrouted: 0
DN:AP371 : | For global routing det
DN:AP152 : | Time: 34
```

Further investigation of the Global Route Ratio Report identifies the bottleneck between fb2.uA and fb1.uD:

\*Traces whose connections are a superset of Connected\_Bins

**QS5.2 AP366 | Global Route Ratio Report**

| Connection          | Trace_Usage | Net_Usage | Ratios |        |        |
|---------------------|-------------|-----------|--------|--------|--------|
|                     |             |           | TDM    | DIRECT | CLOCK  |
| mb_1.uA [-] mb_1.uB | 41/50       | 200       | 0      | 0      | 5      |
| mb_1.uB [-] mb_1.uC | 243/250     | 2163      | 0      | 0      | 9      |
| mb_1.uA [-] mb_1.uC | 243/250     | 2163      | 0      | 0      | 9      |
| mb_1.uA [-] mb_1.uD | 195/200     | 2099      | 1      | 0      | 11     |
| mb_1.uC [-] mb_1.uD | 195/200     | 2218      | 2      | 1      | 11, 12 |
| mb_1.uB [-] mb_1.uD | 198/200     | 2327      | 1      | 0      | 12     |
| mb_1.uC [-] mb_2.uA | 48/50       | 317       | 0      | 0      | 2, 7   |
| mb_1.uB [-] mb_2.uA | 49/50       | 601       | 2      | 0      | 13, 14 |
| mb_1.uD [-] mb_2.uA | 198/200     | 3087      | 2      | 0      | 16     |

### Step 3: Edit Interconnect Definition and Rerun Exploratory Partitioning

To ease the bottleneck, edit the pcf file to increase the interconnect capacity between fb2.uA and fb1.uD from 200 to 300:

```
abstract_tss -add_trace_group {fb_1.uD fb_2.uA} -width 300
```

When run with the increased capacity, the maximum TDM ratio for the design goes down from 16 to 12:

|                   | Report  | Port_Usage | TDM | DIRECT | RATIOS |
|-------------------|---------|------------|-----|--------|--------|
| mb_1.uA[-]mb_1.uB | 43/50   | 115        | 0   | 0      | 2,3    |
| mb_1.uB[-]mb_1.uC | 243/250 | 2405       | 0   | 0      | 10     |
| mb_1.uA[-]mb_1.uC | 244/250 | 2163       | 1   | 0      | 9      |
| mb_1.uA[-]mb_1.uD | 199/200 | 1761       | 1   | 0      | 9      |
| mb_1.uC[-]mb_1.uD | 199/200 | 1895       | 0   | 0      | 10,9   |
| mb_1.uB[-]mb_1.uD | 197/200 | 1853       | 1   | 1      | 10,9   |
| mb_1.uC[-]mb_2.uA | 48/50   | 267        | 1   | 0      | 6      |
| mb_1.uB[-]mb_2.uA | 45/50   | 290        | 2   | 0      | 7      |
| mb_1.uD[-]mb_2.uA | 286/300 | 3258       | 3   | 0      | 11,12  |
| mb_1.uA[-]mb_2.uA | 48/50   | 241        | 2   | 0      | 5,6    |

### Step 4: Transfer Final Results to TSS and Run Partitioning

Use the board\_system\_create -interconnect commands in the TSS file to define the hardware setup that produced the lower TDM ratio. Remove the original abstract\_tss commands from the pcf file, or do not use that pcf file for subsequent partitioning runs once you have a TSS definition with the hardware information. This is a sample of the TSS interconnect commands for fb1A:

```
board_system_create -interconnect -manual CON_CABLE_50_HT3
 -name fb1A9_fb1B1 -connector {fb_1.A9 fb_1.B1}
board_system_create -interconnect -manual CON_CABLE_50_HT3
 -name fb1A10_fb1C23 -connector {fb_1.A10 fb_1.C23}
board_system_create -interconnect -manual CON_CABLE_50_HT3
 -name fb1A11_fb1C22 -connector {fb_1.A11 fb_1.C22}
```

```
board_system_create -interconnect -manual CON_CABLE_50_HT3
 -name fb1A12_fb1C21 -connector {fb_1.A12 fb_1.C21}
board_system_create -interconnect -manual CON_CABLE_50_HT3
 -name fb1A22_fb1C12 -connector {fb_1.A22 fb_1.C12}
board_system_create -interconnect -manual CON_CABLE_50_HT3
 -name fb1A23_fb1C11 -connector {fb_1.A23 fb_1.C11}
board_system_create -interconnect -manual CON_CABLE_50_HT3
 -name fb1A13_fb1D23 -connector {fb_1.A13 fb_1.D23}
board_system_create -interconnect -manual CON_CABLE_50_HT3
 -name fb1A14_fb1D22 -connector {fb_1.A14 fb_1.D22}
board_system_create -interconnect -manual CON_CABLE_50_HT3
 -name fb1A15_fb1D21 -connector {fb_1.A15 fb_1.D21}
board_system_create -interconnect -manual CON_CABLE_50_HT3
 -name fb1A16_fb1D20 -connector {fb_1.A16 fb_1.D20}
board_system_create -interconnect -manual CON_CABLE_50_HT3
 -name fb1A21_fb2A21 -connector {fb_1.A21 fb_2.A21}
```

Additionally, you can set a tdm\_control constraint in the pcf file to fix the ratio and ensure that partitioning and system route are constrained. For example:

```
tdm_control -type ACPM -max_ratio 12
```

## Validating TSS Cable Connections with `conspeed_hstdm`

Use the `conspeed_hstdm` command to validate cable connections and check channel connectivity defined in the TSS file. (HSTDMD is high speed time-domain multiplexing of signals.) The `conspeed_hstdm` command complements the board bring-up `haps con_speed` commands. The validation runs in two phases; implementation and runtime. The first phase, implementation, generates the bit files based on the defined hardware connections. The second phase, runtime, validates the bit files in the hardware setup.

The advantage of the `conspeed_hstdm` command over `haps con_speed` is that you can use it to validate a mixed hardware setup with both HAPS-70 and HAPS-80 systems. Also, it validates the HAPS ProtoCompiler and Vivado version used in conjunction with the TSS file for the given hardware setup. For complete syntax details, see [\*conspeed\\_hstdm\*, on page 25](#) in the *Command Reference Manual*.

In the runtime phase, the command automatically locates the device ID on the host and configures the project with the bit files generated in the implementation flow. It then trains the HSTDMD and validates the data. Finally, it prints the status (Pass/Fail) in the Tcl window and writes the results to the file `hstdm_cable_status.rpt` in the runtime directory.

Pass status validates the cable setup. If the status is Fail, check the setup for loose connections, bent pins, bad cables, etc., then modify the setup as needed before rerunning the validation check. Once you have validated the cable setup with `conspeed_hstdm`, return to your compiled design and go through the normal stages of the design flow.

In the following example, the hardware setup uses multiple hosts, where each host has superchain setups. Host 1 has two superchains, and Host 2 has three superchains, so you will need to provide the host serial number CSV file to the `conspeed_hstdm` command. For this example, the CSV file is `host_slno.txt`.

### CSV File Example

```
Host1, H000001
Host1, H000002
Host2, H000003
Host2, H000004
Host2, H000005
```

The order of HAPS systems listed in the CSV file should match the order of the systems in the TSS file. For example, FB1 should be H000001, FB2 should be H000002, etc.

1. Initialize the setup with the -init and -rundir arguments:

```
conspeed_hstdm -init -rundir myTest
```

With these arguments, the command creates a directory at the specified location with all the files needed to check the connectivity.

2. Generate the bit files with the conspeed\_hstdm command.

Example:

```
conspeed_hstdm -technology HAPS80 -mode bidir
-flow implementation -tss myTss.tcl -rundir myTest
-hstdm_ratio 15 -tdm_type HSTDMD_ERD -bitrate 1400
```

3. Run the bit files and validate the cable setup by doing the following:

- Make sure the hardware setup reflects the definitions in the TSS file.
- Launch Confpro, then source the conspeed\_hstdm command again. This time use the -flow runtime argument and the list of hardware serial numbers of each HAPS system in the superchain.

Example:

```
conspeed_hstdm -mode bidir -flow runtime -serial_numbers {H123456
H654321} -tss myTss.tcl -rundir myTest
```

4. To rerun the validation test, do the following:

- Modify the cable setup and the TSS as needed.
- Clear the current results with the conspeed\_hstdm -clear command.  
  

```
conspeed_hstdm -clear -rundir myTest
```
- Re-run the conspeed\_hstdm command (step 2).

## Validating the TSS Against the Hardware

One of the board bring-up utilities generates a TSS file based on the hardware setup. The following procedure shows you how to generate this file.

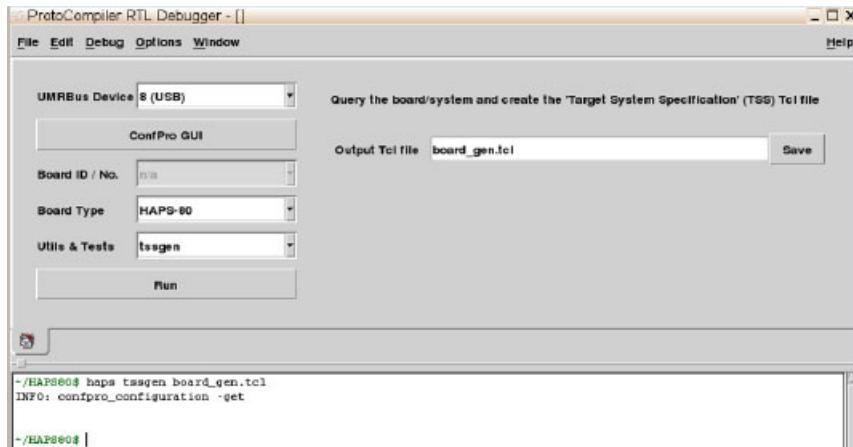
1. Set up the hardware correctly.

2. Start the utility:

```
protocompiler_runtime board_bringup
```

3. Generate a tss file based on the hardware setup by entering this command:

```
haps tssgen board_gen.tcl
```



The command queries the hardware setup and generates a Tcl file with TSS information that reflects the current hardware setup.

4. Use the generated TSS file as a basis for creating an initial TSS file, or to validate that your TSS file matches the hardware setup.

# Generating TSS Files Using TSS Builder

You can either generate a new TSS file or modify an existing TSS file using the TSS Builder.

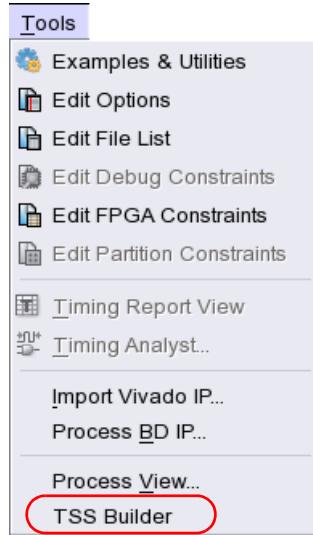
Generating or modifying a TSS file using the TSS Builder involves the following steps:

1. Choose Systems and Daughter Boards, See [\*Choosing Systems and Daughter Boards, on page 229.\*](#)
2. Complete CDE Chaining, See [\*Chaining CDEs, on page 232.\*](#)
3. Assign Clocks, See [\*Assigning Clocks, on page 233\*](#)
4. Assign Top IO, See [\*Assigning Top IO, on page 235.\*](#)
5. Connect Cables within and Across Systems, See [\*Connecting Cables Within and Across Systems, on page 236.\*](#)
6. Connect Daughter Boards, See [\*Connecting Daughter Boards, on page 237.\*](#)
7. Generate TSS File, See [\*Generating TSS Files, on page 238\*](#)

If you are modifying a TSS file that was not created using the TSS Builder, the **-auto** option for `board_system_create -interconnect -manual` cable settings is not supported. However, configuration details are preserved and added to the generated TSS file. If there are references to daughter boards that are not supported by TSS Builder, a warning message is displayed. The references are omitted from the TSS file generated using the TSS Builder. Add back these references manually.

## Choosing Systems and Daughter Boards

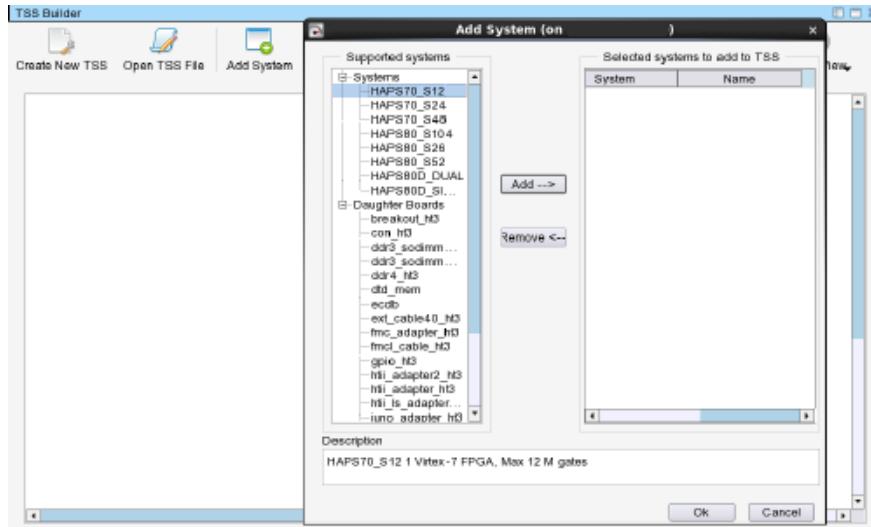
1. Select TSS Builder from the Tools menu to open the TSS Builder Window.



2. Click the Create New TSS File icon to create a new TSS File, or Click Open TSS File icon to open and modify an existing TSS file.

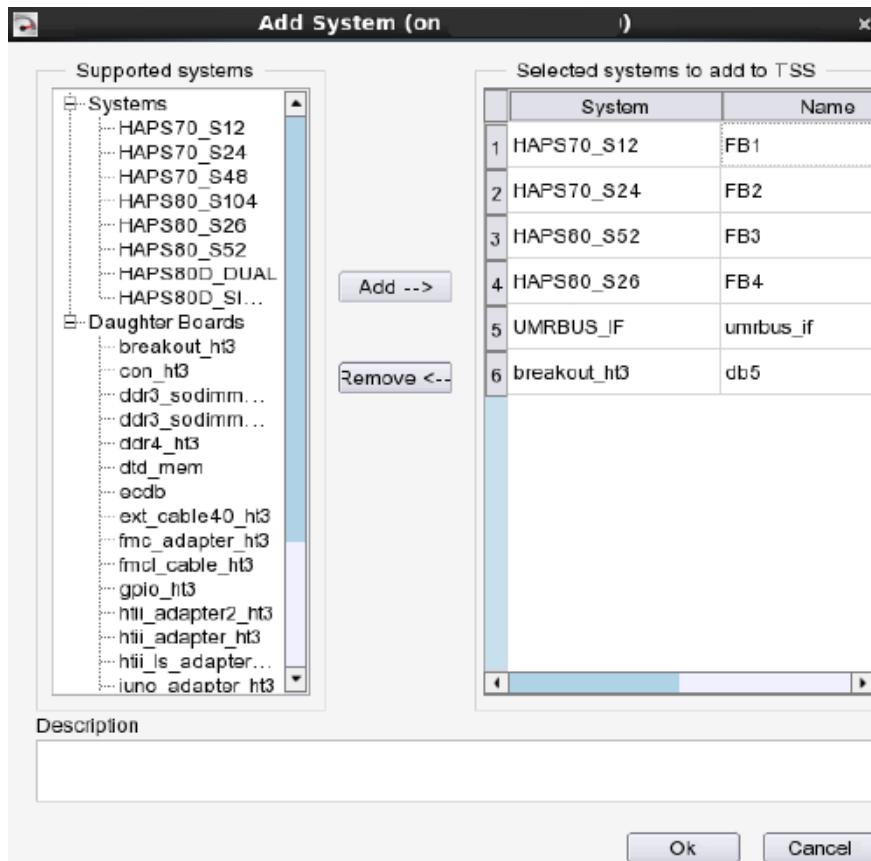
If you choose Creating New TSS File, the TSS Builder Window displays a blank screen.

3. Click Add System icon. The Add System dialog box appears with a list of supported systems and daughter boards.



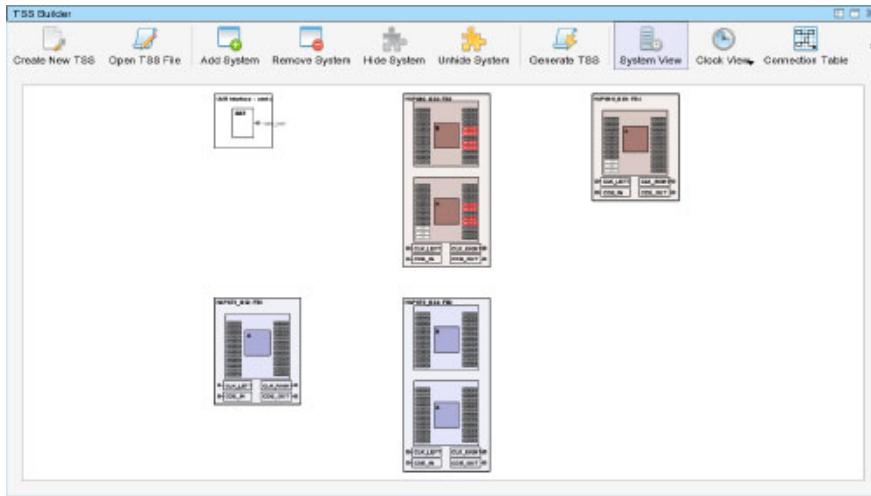
**Note:** Only two HAPS-80D systems can be added to the TSS. You cannot add HAPS-80D with any other system; including the UMRBus interface. For details on supported systems and limitations, see the [TSS Builder, on page 57](#) in the *Reference Manual*.

4. Select the required system from the left pane and click Add. You can select only one system at a time.
5. Select the daughter board from the left pane and click Add. You can select only one daughter board at a time.
6. If you have more than one system in the TSS, add the UMRBus interface from the left pane. This step is not applicable for HAPS-80D.
7. Continue adding all the required systems to the TSS.



All the selected systems and daughter boards will be listed under the Selected systems to add to TSS pane. By default, all these systems and daughter boards are provided with a name. You may change the name and description of the system in the respective fields. To remove a system from the selected list, click the name of the system and click Remove.

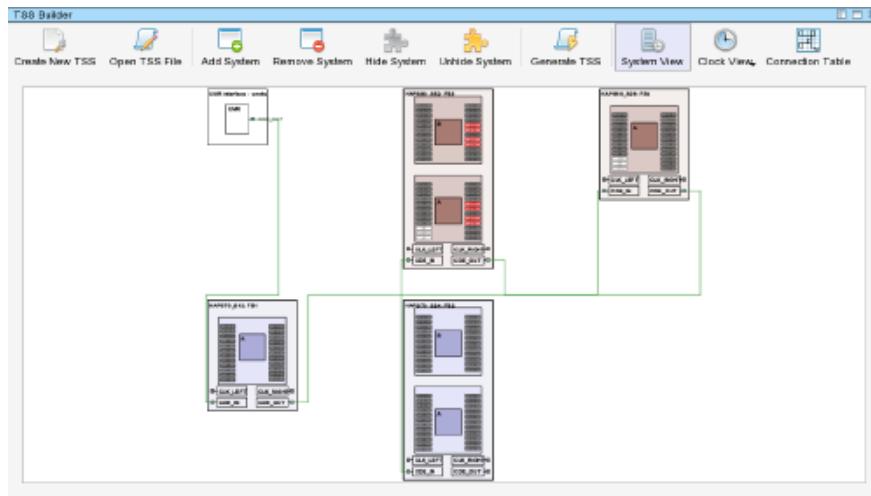
8. Click OK to open System View, which is a graphical representation of the TSS.



## Chaining CDEs

If you have more than one system in the TSS, start the CDE chain from the UMRBus interface (Not applicable for HAPS-80D). All systems must be chained correctly to generate a TSS file.

1. Click the CDE\_OUT port on the UMRBus interface and drag it to the CDE\_IN port of the system to which you want to connect.
2. Click the CDE\_OUT port of the first system and drag it to CDE\_IN of the next system.
3. Continue connecting the CDE\_IN and CDE\_OUT connectors until the CDE connections form a chain.



---

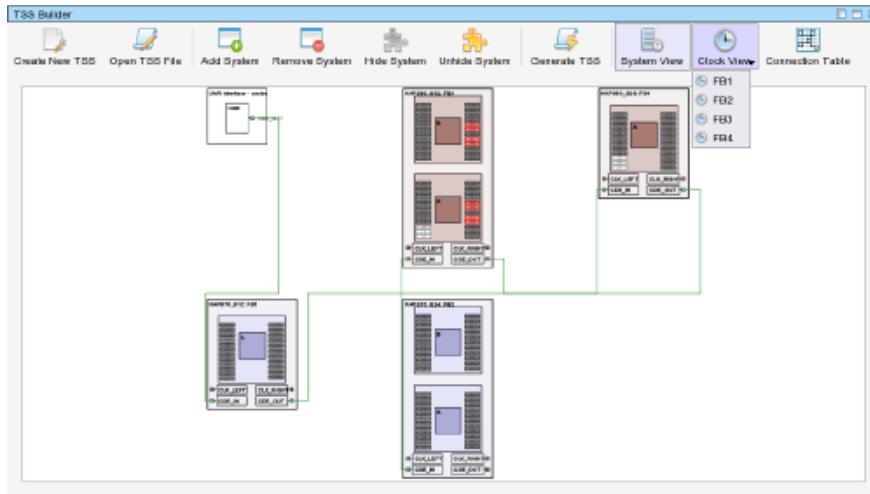
**Note:** Do not configure a Single FPGA as the last system in the CDE chain.

---

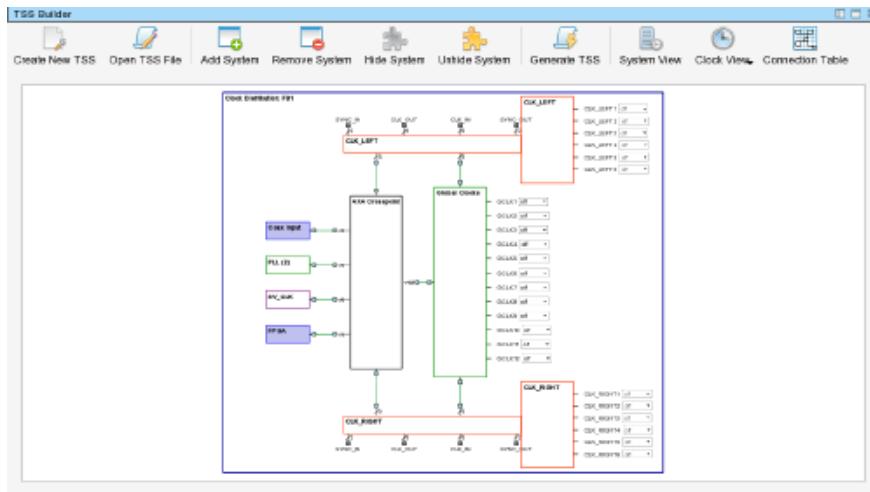
Move your mouse over a connection to see the connection details. To delete a connection, right-click over a connection and select Delete.

## Assigning Clocks

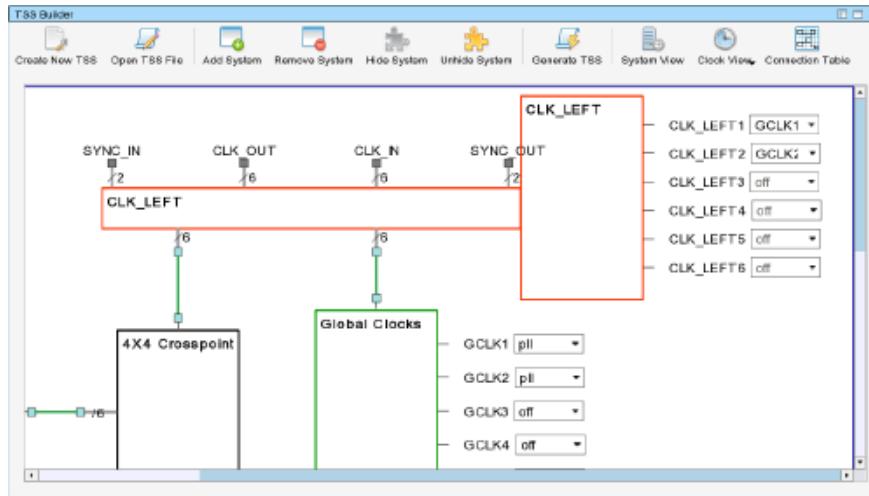
1. Click the Clock View icon and select a system to assign the clocks.



The Clock View for the system appears. By default, all clocks will be in off position.



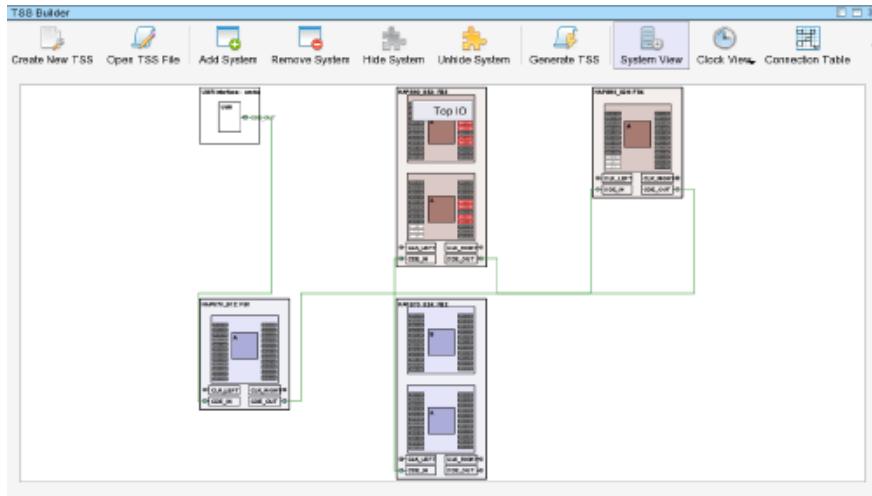
2. Select the clock to be used for the required Global Clock connectors. The corresponding CLK\_LEFT and CLK\_RIGHT clocks are also activated.



3. Select the corresponding CLK\_LEFT or CLK\_RIGHT from the drop-down list.
4. Continue assigning the clocks as required.
5. Click the System View icon to return to the System View.

## Assigning Top IO

Before you connect the systems, you must set up at least one Top IO in the TSS. Right-click on any connector and select Top IO from the pop-up menu.



## Connecting Cables Within and Across Systems

1. Click the desired connector on a system and drag it to the connector to which you want to connect it to.
2. Repeat connecting the connectors until all required connections are made.



By default, all connections are made using a CON\_CABLE\_25\_HT3 cable. To modify the length of the cables, right-click the cable and select the cable length from the pop-up menu. Cables are color coded based on length. If you have several connections, you can use the Zoom In option to take a closer look. You can hide systems, using the Hide option. For details, see [Hide System, on page 62](#) in the *Reference Manual*.

---

**Note:** On a HAPS-80 System, you cannot configure connections for connectors A1, A2, and A3. On a HAPS-80D, you cannot configure connections for connectors A1, A2, A3, and A12. TSS Builder displays these connectors in white.

---

3. (Optional) You can also view all connections and add new connections using the Connection Table. Click the Connections Table icon to open the table.

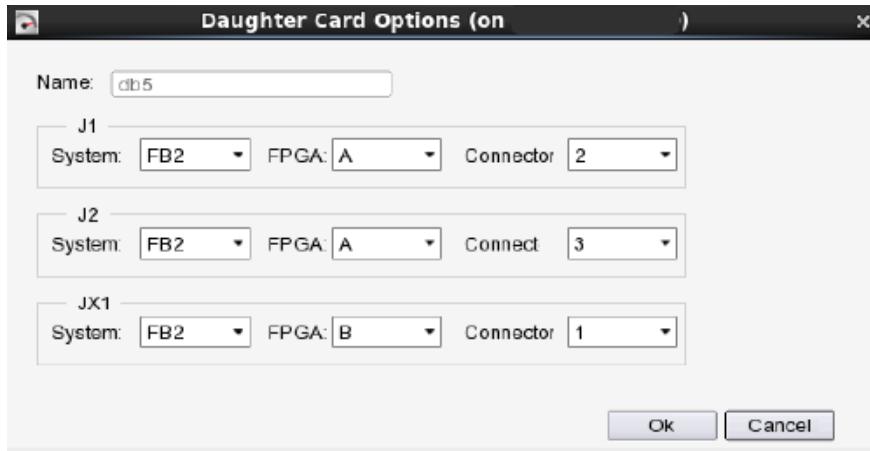
| Name              | Port 1  | Port 2  | Cable Used        |
|-------------------|---------|---------|-------------------|
| 1 FB1.A20_FB3.B6  | FB1.A20 | FB1.B8  | CON_CABLE_25_HT3  |
| 2 FB3.A24_FB4.A4  | FB3.A24 | FB4.A4  | CON_CABLE_50_HT3  |
| 3 FB3.B13_FB4.A12 | FB3.B13 | FB4.A12 | CON_CABLE_100_HT3 |
| 4 FB1.A23_FB3.B6  | FB1.A23 | FB3.B6  | CON_CABLE_100_HT3 |
| 5 FB1.A12_FB3.A12 | FB1.A12 | FB3.A12 | CON_CABLE_100_HT3 |
| 6 FB1.A10_FB2.A1  | FB1.A10 | FB2.A1  | CON_CABLE_150_HT3 |
| 7                 |         |         |                   |

## Connecting Daughter Boards

1. Click the Daughter Board Table icon. A table with a list of daughter boards added to the system appears.

| Daughter Board     | Device Name | Connected to System | Ports           | Notes                  |
|--------------------|-------------|---------------------|-----------------|------------------------|
| 1 BREAKOUT-HT3.dbs | NONE        |                     | J1<br>J2<br>JX1 | No ports are connected |
| 2                  |             |                     |                 |                        |

2. Double-click the required daughter board to open the Daughter Card Options dialog box.



3. Select the required system, FPGA, and connector to which each port on the daughter board is to be connected.
4. Click OK. The Daughter Board table is updated with the connection details.
5. Click the System View icon to return to the System View.

## Generating TSS Files

1. Click Generate TSS icon to open the TSS file in a TSS Configuration Script window.

```

board_system_create -haps -name DefaultSystem

board_system_create -add HAPS70_S12 -name FB1
board_system_create -add HAPS70_S24 -name FB2
board_system_create -add HAPS80_S52 -name FB3
board_system_create -add HAPS80_S26 -name FB4
board_system_create -interconnect -manual BREAKOUT_HT3 -name db5 -connector {FB2.A2 FB2.A3 FB2.B1}

board_system_configure -top_lo {FB3.B12}

board_system_configure -clock FB1.GCLK1 pll
board_system_configure -clock FB1.GCLK10 pll
board_system_configure -clock FB1.GCLK11 pll
board_system_configure -clock FB1.GCLK12 pll
board_system_configure -clock FB1.GCLK2 pll
board_system_configure -clock FB1.GCLK5 pll
board_system_configure -clock FB1.GCLK6 pll
board_system_configure -clock FB1.GCLK7 pll
board_system_configure -clock FB1.GCLK8 pll
board_system_configure -clock FB1.GCLK9 pll
board_system_configure -clock FB1.CLK_LEFT1 pll
board_system_configure -clock FB1.CLK_LEFT2 pll
board_system_configure -clock FB1.CLK_LEFT3 pll
board_system_configure -clock FB1.CLK_LEFT6 pll
board_system_configure -clock FB1.CLK_RIGHT1 pll
board_system_configure -clock FB1.CLK_RIGHT2 pll
board_system_configure -clock FB1.CLK_RIGHT3 pll
board_system_configure -clock FB1.CLK_RIGHT4 pll
board_system_configure -clock FB1.CLK_RIGHT5 pll
board_system_configure -clock FB1.CLK_RIGHT6 pll
board_system_configure -clock FB2.GCLK1 pll
board_system_configure -clock FB2.GCLK4 left
board_system_configure -clock FB2.CLK_LEFT1 pll
board_system_configure -clock FB2.CLK_LEFT4 pll

board_system_configure -connect {unibus_if.CDE_OUT FB1.CDE_IN}
board_system_configure -connect {FB1.CDE_OUT FB3.CDE_IN}
board_system_configure -connect {FB3.CDE_OUT FB4.CDE_IN}
board_system_configure -connect {FB4.CDE_OUT FB2.CDE_IN}

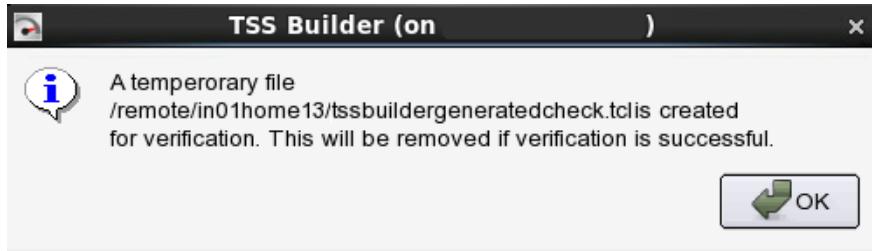
Total number of con_cables used are as follows :
CON_CABLE_HT3_25: 1
CON_CABLE_HT3_50: 1
CON_CABLE_HT3_100: 2
CON_CABLE_HT3_150: 1
CON_CABLE_HT3_200: 1

board_system_create -interconnect -manual CON_CABLE_25_HT3 -name FB1.A20_FB3.B8 -connector {FB1.A20 FB3.B8}
board_system_create -interconnect -manual CON_CABLE_50_HT3 -name FB3.A24_FB4.A4 -connector {FB3.A24 FB4.A4}
board_system_create -interconnect -manual CON_CABLE_100_HT3 -name FB3.B13_FB4.A12 -connector {FB3.B13 FB4.A12}
board_system_create -interconnect -manual CON_CABLE_200_HT3 -name FB1.A23_FB3.B6 -connector {FB1.A23 FB3.B6}
board_system_create -interconnect -manual CON_CABLE_100_HT3 -name FB1.A12_FB3.A12 -connector {FB1.A12 FB3.A12}
board_system_create -interconnect -manual CON_CABLE_150_HT3 -name FB1.A10_FB2.A1 -connector {FB1.A10 FB2.A1}

```

- Click Verify to check the TSS file for errors.

A temporary file `tssbuildergeneratedcheck.tcl` is created for verification. This file is removed when verification is successful. A result window appears after the verification is complete.

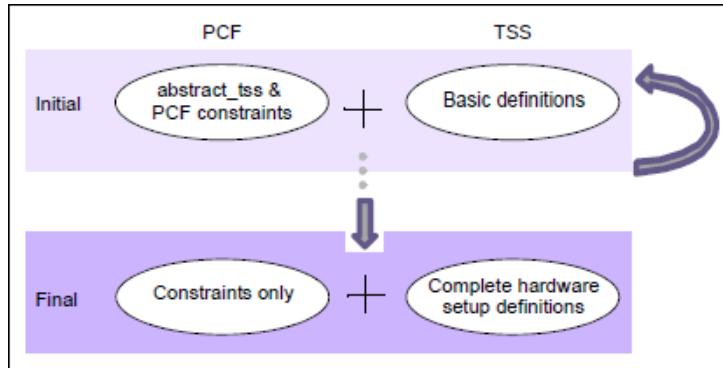


3. Click Save or Save As to save the TSS file.

# Defining Constraints in PCF Files

The Partitioning Constraints Format (PCF) file ensures greater accuracy and more meaningful results by providing a mechanism to specify constraints that guide the auto-partitioner during partitioning.

Partitioning is iterative, and you can use the PCF file in different ways, depending on the state of the design. It is recommended that you create multiple pcf files so that you can use them for different purposes in a modular fashion.



Note that the pcf file is used for two different purposes:

- To create abstract interfaces and interconnect schemes for initial partitioning and exploration.
- To define assignment constraints to guide actual partitioning and system routing, including constraints for bin utilization, floorplanning, hierarchy control, and assignments of cells and ports. Bins are groups of objects created by the partitioner.

This table summarizes where to find information about using the pcf file.

| For Information About ...                                                                                                                                                                                               | See ...                                                                                                                                                                                                                                     |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>Partition exploration with the pcf file</li> <li>Using the pcf file to develop specifications for the tss file</li> <li>Using abstract pcf constraints</li> </ul>                | <a href="#">Using Abstract PCF Commands for Partition Exploration</a> , on page 242                                                                                                                                                         |
| <ul style="list-style-type: none"> <li>Getting tss information</li> <li>Querying the design</li> </ul>                                                                                                                  | <a href="#">Using PCF Queries for TSS and Netlist Information</a> , on page 261                                                                                                                                                             |
| <ul style="list-style-type: none"> <li>Setting partition assignment constraints for clocks, ports, and cells</li> <li>Setting assignment constraints to guide run partition</li> <li>Manual trace assignment</li> </ul> | <a href="#">Specifying PCF Constraints for Partitioning</a> , on page 245<br><a href="#">Specifying PCF Constraints in the PCF Editor (GUI)</a> , on page 254<br><a href="#">Manually Assigning Inter-FPGA Nets to Traces</a> , on page 265 |
| <ul style="list-style-type: none"> <li>Moving from abstract to detailed partitioning</li> <li>Iterating between pre-partition and partitioning runs with the pcf</li> </ul>                                             | <a href="#">Moving from Abstract PCF to TSS: Flow Example</a> , on page 220<br><a href="#">Working Iteratively with PCF Assignments</a> , on page 267                                                                                       |

## Using Abstract PCF Commands for Partition Exploration

If you are in the early stages of prototyping, you experiment with various interconnect schemes before committing to one. For example, you might know you need four FPGAs and 500 ports, but not have any other design restrictions. Use pcf constraints to explore partition choices and determine the final target hardware configuration.

When you are exploring partitioning options, use the pcf file and abstract constraints to specify the interconnect for an abstract target system definition, as described here. For details about the pcf constraint syntax, see [Partition Constraint File Tcl Commands](#), on page 165 in the *Command Reference Manual*.

1. Do a baseline partitioning run (run partition).
  - After the run, check the partitioning report and check the number of FPGAs and port bins.
  - Check the report (report target\_system) for global clock names.

2. Create a pcf file with constraints that define the I/O interface for the FPGAs, based on the FPGA names from the initial partitioning run.
  - Get the names of the FPGA bins by checking the TSS report ([Checking the Target Specification \(TSS\), on page 358](#))
  - Use the abstract\_tss -add\_port\_bin command to define the interface:

```
abstract_tss -add_port_bin -fpga FB1.uA -name \ ABS_PORTA -width 650
abstract_tss -add_port_bin -fpga FB1.uB -name \ ABS_PORTB -width 650
abstract_tss -add_port_bin -fpga FB1.uC -name \ ABS_PORTC -width 650
abstract_tss -add_port_bin -fpga FB1.uD -name \ ABS_PORTD -width 650
```

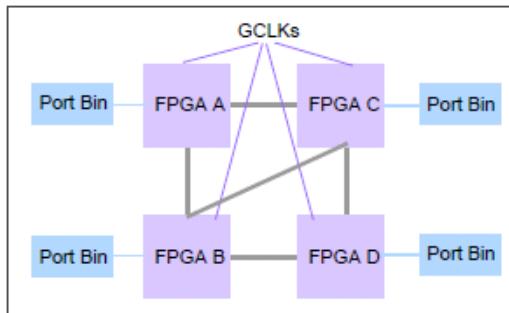
  - Set port bin sizes with the -width argument. You can expand the size of abstract ports beyond the connector size, to provide the tool with more freedom during this exploratory phase. The example above increases the width to 650, even though one HT3 connector has just 50 pins.

If you have existing tra and prt constraints that you want to convert to pcf constraints, contact Synopsys.

3. Define FPGA-to-FPGA link pairs to explore interconnect schemes, using the abstract\_tss -add\_trace\_group command.

This example shows a diagonal interconnect scheme for four FPGAs:

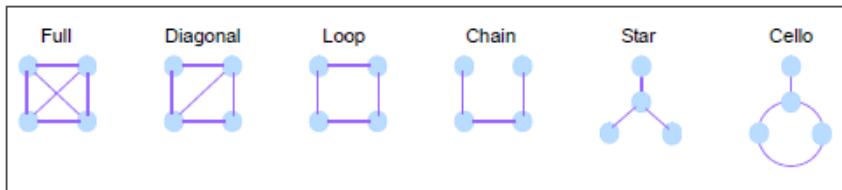
```
abstract_tss -add_trace_group {FB1.uA FB1.uB} -width 500
abstract_tss -add_trace_group {FB1.uB FB1.uD} -width 500
abstract_tss -add_trace_group {FB1.uD FB1.uC} -width 500
abstract_tss -add_trace_group {FB1.uC FB1.uA} -width 500
abstract_tss -add_trace_group {FB1.uC FB1.uB} -width 500
```



Here are some guidelines for defining interconnect:

- Target the smallest max ratio with the fewest cables

- Explore other hardware topologies. The following figure shows the six hardware topologies possible for 4-FPGA systems. The Star and Chain configurations use the fewest cables.



- Use inter-FPGA feedthroughs if necessary. Inter-FPGA feedthroughs are routed without pin multiplexing, and are usually faster than a pin-multiplexed route.
- Avoid port feedthroughs.

If your TSS file defines port bins, external bins, and interconnect like clocking architecture, use related pcf commands that modify the defined target system for exploration. For example, you can use pcf commands to assign clocks if your TSS file defines inter-FPGA clocks.

- Iteratively run the partitioner and adjust the pcf constraints until you have a feasible interconnect scheme, and meet your minimum performance target.
- You can now create a detailed tss that implements the abstract interconnect scheme, and use it for partitioning and system routing.

When you switch to a detailed tss, the pcf file must not contain any abstract interconnect commands. All pcf constraints must be partitioning constraints related to clock assignments, cell assignments, port assignments, and so on. See the following for more information:

Setting final pcf constraints: clock, cell, and port assignments

[Specifying PCF Constraints for Partitioning, on page 245](#)

[Specifying PCF Constraints in the PCF Editor \(GUI\), on page 254](#)

Example illustrating the flow from pcf to tss

[Moving from Abstract PCF to TSS: Flow Example, on page 220](#)

TSS file

[Defining the Target System in a Detailed TSS File, on page 189](#)

Partitioning flow

[Design Methodology for Partitioning, on page 179](#)

## Specifying PCF Constraints for Partitioning

The PCF constraints described here are not abstract interconnect constraints used during the exploration phase ([Using Abstract PCF Commands for Partition Exploration, on page 242](#)), but specific constraints for port, clock, and cell assignments. Not all the pcf constraints are described; for a complete list of all the pcf constraints and their syntax, refer to [Partition Constraint File Tcl Commands, on page 165](#) in the *Command Reference Manual*.

### Defining PCF Partitioning Constraints

The procedure uses commands to specify the constraints, but alternatively you can use the GUI editor to create a pcf file ([Specifying PCF Constraints in the PCF Editor \(GUI\), on page 254](#)).

1. Create or update the pcf file from the pre-partition, partition, or system route states.

Make sure to update the pcf file after partition, so that you can use the results of the partitioning run to define more accurate constraints.

It is recommended that you use multiple PCF files for different constraints. See [Using Multiple PCF Files, on page 246](#).

2. Enter the constraints you need.

Global setup      [Defining Global Information in the PCF, on page 246](#)

Port assignments      [Assigning Ports in the PCF, on page 247](#)

Cell assignments      [Assigning Cells in the PCF, on page 249](#)

Clock assignments      [Assigning Clocks in the PCF, on page 251](#)

Nets for MGBLINKS      [Assigning Nets to MGBLINKS, on page 254](#)

3. Check the partitioning report and partitioning log files for results. See [Analyzing Partition Result Files, on page 358](#) for more information.

### TDM on Asynchronous Nets

You can add TDM on an asynchronous net using the following PCF switches:

```
partition_optimization_control -relax_async_constraints 1
```

```
report_control -async_nets 1
```

The tool will automatically detect the asynchronous nets in which the source and destination flip-flops are in different clock domains. To use these options set the license\_feature, FpgalInternalBeta. See [Target System Definition, on page 167](#) for more information.

## Using Multiple PCF Files

It is recommended that you create multiple pcf files with different constraints so that you can use them in a modular fashion. Here are some suggested pcf files you might use at different points in the design cycle:

|                 |                                                                                                                                                                                                |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| abstract.pcf    | FPGA definitions and interconnect schemes defined with abstract_tss commands for design exploration. See <a href="#">Using Abstract PCF Commands for Partition Exploration , on page 242</a> . |
| setup.pcf       | Global prototype information, like bin utilization, TDM controls, and cell dissolving. See <a href="#">Defining Global Information in the PCF , on page 246</a> .                              |
| ports.pcf       | Port assignments. Initially, specify them as clusters or assign them to bins. Later, assign them to pins or traces. See <a href="#">Assigning Ports in the PCF , on page 247</a> .             |
| clocks.pcf      | Clock assignments. See <a href="#">Assigning Clocks in the PCF , on page 251</a> .                                                                                                             |
| floorplan.pcf   | Assignment of a few key cells, specific bin assignments and set assignments. See <a href="#">Assigning Cells in the PCF , on page 249</a> .                                                    |
| assignments.pcf | Assignment of all cells when you want to lock down a partition. These assignments can be exported out of the tool. See <a href="#">Assigning Cells in the PCF , on page 249</a> .              |

## Defining Global Information in the PCF

This procedure shows you how to define global setup information using PCF constraints. Global information includes specifications for bin utilization, TDM controls, and cell dissolving, and it is recommended that you put this kind of information in setup.pcf. The following procedure describes some of the commands, but is not an exhaustive list of the options. Consult [Partition Constraint File Tcl Commands, on page 165](#) in the *Command Reference* for a complete list.

1. Define resource availability and set maximum utilization constraints with the `bin_utilization` command:
  - Check resources with the `design_list_cells -filter` command.
  - To define bin utilization as a ratio, use `bin_utilization -resource_ratio`:

```
Use 80% of the available resources for all bins
bin_utilization -all_bins -resource_ratio {All 0.8 }

Use 90% of LUT resources for FPGA bin FB1.uA
bin_utilization -bin FB1.uA -resource_ratio {LUT 0.9}
```
  - To define bin utilization as a number value, use `bin_utilization -resource`:

```
Use up to 100,000 LUT resources for FPGA bin FB1.uB
bin_utilization -bin FB1.uB -resource {LUT 100000}
```

For syntax details, see [bin\\_utilization, on page 180](#) in the *Command Reference*.

2. For controls on individual cells use `cell_attribute -dissolve`, as described in [Assigning Cells in the PCF, on page 249](#).
3. Set options for TDM with `tdm_control` commands.

See [Using Time Domain Multiplexing \(TDM\), on page 372](#) for details.

## Assigning Ports in the PCF

Initially, specify ports as clusters or assign them to bins. As the design progresses, assign them to pins or traces. You can put this information in a separate file, `ports.pcf`. The following procedure describes some of the commands, but is not an exhaustive list of the options. Consult [Partition Constraint File Tcl Commands, on page 165](#) in the *Command Reference* for a complete list.

1. Assign top-level ports to legal port bins with the `assign_port` command.

You can also use this command to specify a particular pin for a top-level port.
2. Assign clock and reset ports to special bins.
3. Group ports for logical interfaces together with the `cluster_port` command.

The following example defines ports that must be kept together because they define an external interface:

```
cluster port {addr[12:0] data[15:0] r_not_w strobe}
```

4. If you want the automatic partitioner to assign ports and ignore assignments made with `define_haps_io`, you must specifically set `port_attribute` in the pcf file so that the partitioner ignores `define_haps_io`:

```
port_attribute -ignore [portBusNameList]
```

If this attribute is set, the tool ignores `define_haps_io` assignments and uses the assignments made by the partitioner.

5. Optionally, assign nets with the `cluster_net` command.

Use net assignments to group nets if you do not get the results you want with just cell and port assignments. With this command, the partitioner keeps the drivers of clustered nets together. You can assign a name to the cluster; the name can then be used in subsequent net commands to refer to the whole group.

6. Use `net_attribute` commands to assign nets to functional trace groups.

This constraint specifies attributes on nets, which the router later routes automatically, using functional group traces that match the attributes. To manually route nets through specified bins, use the `net_route` command. The following are some typical uses of the `net_attribute` constraint:

- The HAPS-70 system has five buried traces (BC,CD,BD) that are fixed. Assign the nets to a special trace group for fixed traces with this command:

```
net_attribute {netNameList} -function five_fixed_traces
```

- For HAPS-70 systems, assign GCLK8 to a special trace group:

```
net_attribute {netNameList} -function GCLK8
```

- Use the constraint to specify nets for later automatic assignment to MGBLINKs by the router.

```
net_attribute {netNameList} -function MGT
```

For details about manual assignment, see [Assigning Nets to MGBLINKs, on page 254](#).

- For HAPS-70 systems, use `net_attribute` along with the `tdm_control -hstdm_reset_trace` option to assign a global reset as the reset for HSTDMD circuit training for HAPS-70 systems. See [Using HSTDMD and HSTDMD ERD, on page 376](#)for details.

```
assign_port {rst} {fb1.reset}
net_attribute {rst} -function {RESET}
tdm_control -type HSTDMD -max_ratio 8 -hstdm_reset_trace fb1.RESETn
```

For an overview of HAPS-80 reset and HSTDMD, see [Using HAPS-Controlled HSTDMD Training, on page 380](#). For details about HAPS-80 features, refer to the documentation.

7. Specify routing constraints for the partitioner and system router.
  - Use `trace_group_attribute` to define trace groups.
  - Remove traces from consideration by the partitioner or router with `reserve_trace`.
8. If you need to manually assign inter-FPGA nets to traces, use the `net_route` command for nets, and the `assign_port` command for daughter boards.

It is recommended that you use the automatic partitioner for trace assignment. For details and examples of manual assignment, see [Manually Assigning Inter-FPGA Nets to Traces, on page 265](#).

## Assigning Cells in the PCF

This procedure shows you how to floorplan your design, using PCF constraints to guide logic assignments for ports and key cells. It is recommended that you put this kind of information in `floorplan.pcf` during the initial stages, and then `assignments.pcf` as the design is finalized and more detailed constraints added. You can export `assignments.pcf`.

The following procedure describes some of the commands, but is not an exhaustive list of the options. Consult [Partition Constraint File Tcl Commands, on page 165](#) in the *Command Reference* for a complete list.

1. Restrict logic assignment to specific FPGAs by locking them with the `bin_attribute -locked` constraint.

No logic is assigned to locked FPGA bins. For example, the following command only assigns logic to FPGAs A and B, because C and D are locked:

```
bin_attribute -locked fb0.uC
bin_attribute -locked fb0.uD
```

## 2. Assign legal bin locations for key cells with the assign\_cell command.

You can assign a cell to multiple bins. However, if a bin is locked, you can only specify a single bin. The tool can cluster assigned cells with other cells that have the same assignment. Some assign\_cell commands create dissolve commands as a side effect.

- For large functional blocks like CPUs that span multiple FPGAs, define a multi-FPGA region for the group of cells. The following examples confine the core cells to three FPGA bins each:

```
floorplan.pcf
assign_cell {dut_mod_inst.sub0.core0} {FB1.uA FB1.uB FB1.uC}
assign_cell {dut_mod_inst.sub1.core1} {FB2.uB FB2.uC FB2.uD}
```

- Only assign black boxes to black box bins.

## 3. Define logic that must be kept together with the cluster command.

Use these commands to group cells that should stay together for timing reasons, or to keep logical interfaces and associated logic together. The tool assigns clusters to the same FPGA or port bin.

- Define hard cell clusters that must be assigned to the same bin:

```
cluster {dut_mod_inst.sub0.blkA dut_mod_inst.sub0.blkB}
```

- Use the -soft argument to indicate clusters that can be split if this leads to better results:

```
cluster {dut_mod_inst.sub1.blkC dut_mod_inst.sub1.blkD} -soft
```

## 4. You can specify dissolve controls on individual cells. In general, leave the default automatic setting for cells (cell\_attribute -dissolve auto), where the tool chooses which design hierarchies to dissolve when a block exceeds the FPGA bin resources. Use other arguments to override the default.

Specify hierarchical logic that can be dissolved and freely assigned to different bins with cell\_attribute *{list of cells}* -dissolve 1 commands. Use report schematic to identify modules that contain mainly feedthroughs and add dissolve constraints for them. You can also use this command for blocks that are primarily made up of random logic.

If the automatic partitioner takes many hours to run, this could be because too many cells are being dissolved. You can set `cell_attribute -dissolve 0` to mark self-contained interfaces that you want to keep intact.

- ```
# Do not dissolve this cell
cell_attribute {asdf.basdf.basdf} -dissolve 0
```
- Check the partitioning report to see which cells were dissolved. For details of the dissolving process, see [Auto Hierarchy Dissolve Controls, on page 208](#) in the *Command Reference Manual*. Check the Bin Usage Summary section of the partitioning report for the cell count, and the Dissolving large cells and must-dissolve cells section to identify which cells are being over-dissolved. Then apply `cell_attribute -dissolve 0` to these cells.
 - 5. Replicate clock buffers in each FPGA with the `cell_attribute -auto_replicate 1` command.
 - To automatically replicate cells, use the `cell_attribute {list of cells} -auto_replicate 1` command.
- Although it is best to avoid replicating MMCMs because replication could result in cut clocks and cause hold violations and hardware failures, you might have to do this if MMCM clocks drive most of your partitioned logic. In this case you can manually replicate the MMCMs based on your knowledge of the design, but also make sure to connect MMCM-generated clocks to a GCLK that connects to all the FPGAs in the design. This helps avoid phase differences in clocks that connect to all the FPGAs in the design. For an example, refer to SolvNet article 2345509.
- Use the `-force` option to override the default and ensure that the cell is replicated even if it does not have connections in the specified bin after partitioning.
 - 6. Customize what is reported in the log file with the `report_control` command.

Assigning Clocks in the PCF

This is a generic procedure that shows you how to use PCF constraints to add clock and reset information to your design; specifics vary with the hardware used.

It is recommended that you use a separate file for clock constraints called `clocks.pcf`. The following procedure describes some of the commands, but is not an exhaustive list of the options. Consult [Partition Constraint File Tcl Commands, on page 165](#) in the *Command Reference* for a complete list.

Be aware that there are some clock PCF command differences for HAPS-100, see [Partition Constraint File Tcl Commands, on page 165](#) in the *Command Reference*

1. Get global clock and other names.
 - Do an initial partitioning run.
 - Use the `report target_system -tss` and `view report` commands to generate reports. Check that bins were created according to the commands you entered.
 - Get the names for global clock nets, resets, and clock bins from this report. Use these names when you declare the system clocks in the next step.
2. Assign clock and reset ports to special bins with the `assign_port` command.
 - Assign global clocks to the appropriate bins with the `assign_port` command. Global clocks can be driven by different sources: PLL, FPGA, or external. In the example below, the `assign_port` commands assign the global clock to the PLL source driver bins, and the `assign_global_net` commands assign the global clock traces.

```
# GCLK1
net_attribute gclk1_p -function GCLK -differential \
    gclk1_n -is_clock 1
# Assign global clock to the pll1 bin
assign_port gclk1_p FB1.clk pll1
assign_port gclk1_n FB1.clk pll1
# Assign global clock trace
assign_global_net gclk1_p FB1.GCLK1

# GCLK5
net_attribute clk25_p -function GCLK -differential \
    clk25_n -is_clock 1
# Assign global clock to the pll2 bin
assign_port clk25_p FB1.clk pll2
assign_port clk25_n FB1.clk pll2
# Assign global trace
assign_global_net clk25_p FB1.GCLK5
```

```
# GCLK8
assign_port gclk8 FB1.clk.bd_clk8
assign_global_net gclk8 FB1.GCLK8
```

- Assign a port to the GCLK0 bin. GCLK0 is the reserved system clock and is input to all FPGAs with a fixed frequency of 100 MHZ.
 - See [Assigning Internally Generated Clocks to the GCLK Network, on page 272](#) for details about assigning internally generated clocks.
3. Assign global clocks to global clock traces.

- Find trace names for the global clocks from the reports described in the first step. This example shows that global clock bin FB1.clk pll1 drives global clock trace FB1.GCLK1:

```
trace FB1.GCLK1 -function {GCLK hard} -from FB1.clk pll1 -pins {
```

- Assign traces for the known global clocks, using the identified trace names with the assign_global_net command. For example:

```
assign_global_net xck_x1 FB1.GCLK1
```

- Provide information about whether the clock has a single port or is differential, with the net_attribute command.

```
net_attribute {xck_x1} -function GCLK -diffsingle -is_clock1
```

- Assign resets to the global reset, as shown below:

```
assign_port xxclr FB1.reset
net_attribute xxclr -function RESET
```

Example of a PCF File

```
Manual cell assignment
#####
# Fuctional group and Trace assignment for external clock source
#####
# net_attribute {clk} -function {GCLK} -replicate 0
# assign_global_net {clk} {mb1.GCLK3}
```

```
##### END Global_Nets
##### BEGIN Port_Assigns - (Populated from PCF Editor, do not
edit)
assign_port {clk} -trace {mb1.GCLK3}
assign_port {clk_out10} -trace {mb1.CLK_SRC[2]}
assign_port {clk_out5} -trace {mb1.CLK_SRC[1]}
assign_port {reset} -trace {mb1.A_USER_RESETN}
```

Assigning Nets to MGBLINKs

You can mark nets and let the tool automatically assign them to MGBLINKs, or you can assign them manually.

1. To mark nets for automatic assignment, specify the function of the nets you want to assign to MGBLINKs as MGT.

```
net attribute netName -function MGT
```

During system route, the tool automatically assigns nets marked with the MGT function to MGBLINKs.

2. To manually assign nets, do the following:

- Use the `report target_system -tss` command to identify the usable MGBLINKs. Search for the traces that have the MGT hard function.
- Define the net assignment with the `net_route` command, using the MGBLINK name you just identified in the `-using_trace` argument.

```
net route {myNet} -from {FB1.uA} -to {FB1.uB} -using_trace
{FB1.MGBLINK_BI2_TO_AI2_FIXED_P[7:0]}
```

Specifying PCF Constraints in the PCF Editor (GUI)

The following procedure describes how to use the GUI PCF editor to enter PCF constraints. This method is an alternative to manually entering the PCF commands in a file, as described in [Specifying PCF Constraints for Partitioning, on page 245](#).

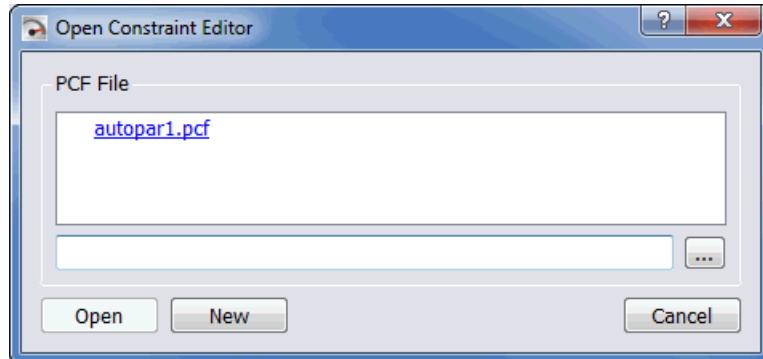
You can also use the editor to edit an existing pcf file.

1. Open the PCF editor.

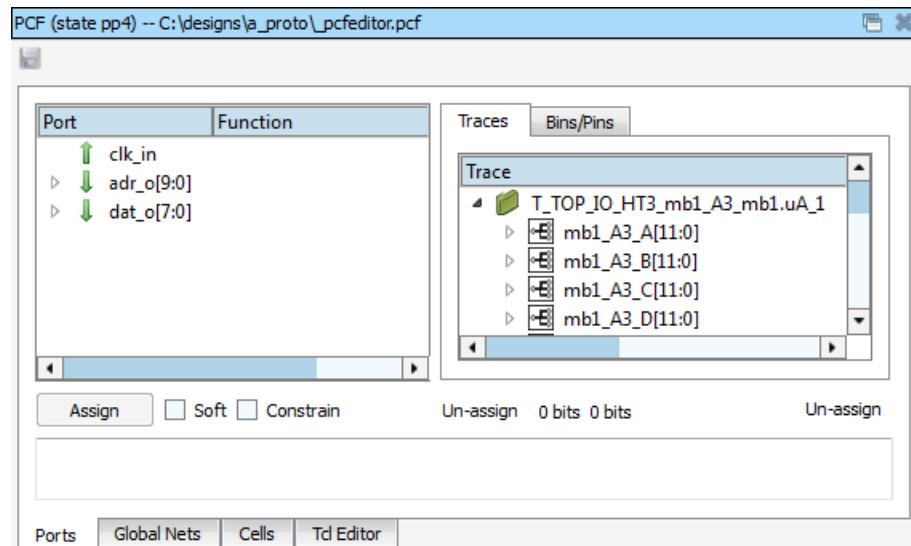
- From an active pre-partition, partition, or system route database state, click the PCF Editor icon () in the top bar of the tool window or select Tools->Edit PCF.

The right side of the tool window opens a new tab with the PCF editor.

- Select a PCF file from the list displayed and click Open, or click New to create a new file.



The PCF graphical editor window opens. The editor window lists design objects in the left pane, which you can assign to target objects in the right pane.

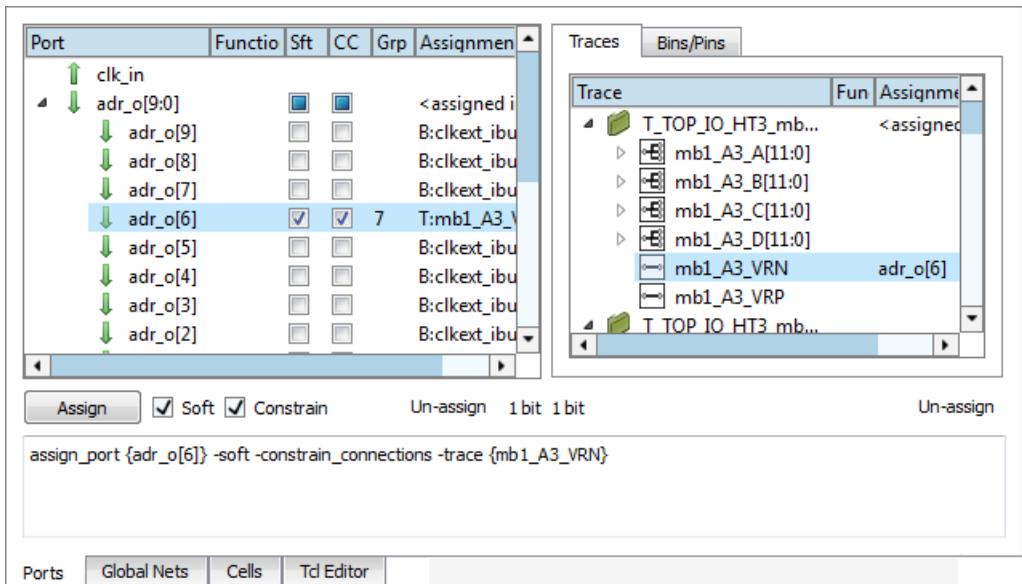


2. To assign a port, follow these steps:

- Go to the Ports tab (below the left pane) and select the I/O port to be assigned from the left pane.
- In the right pane, select the Traces or Bins/Pins tab, as appropriate, and select the object to which you want to assign the selected port.
- Optionally, specify limits for the constraint. To allow the partitioner to remove the constraint to improve results, click the Soft checkbox located next to the Assign button. The tool updates the corresponding Sft check box in the left panel when you assign the ports.

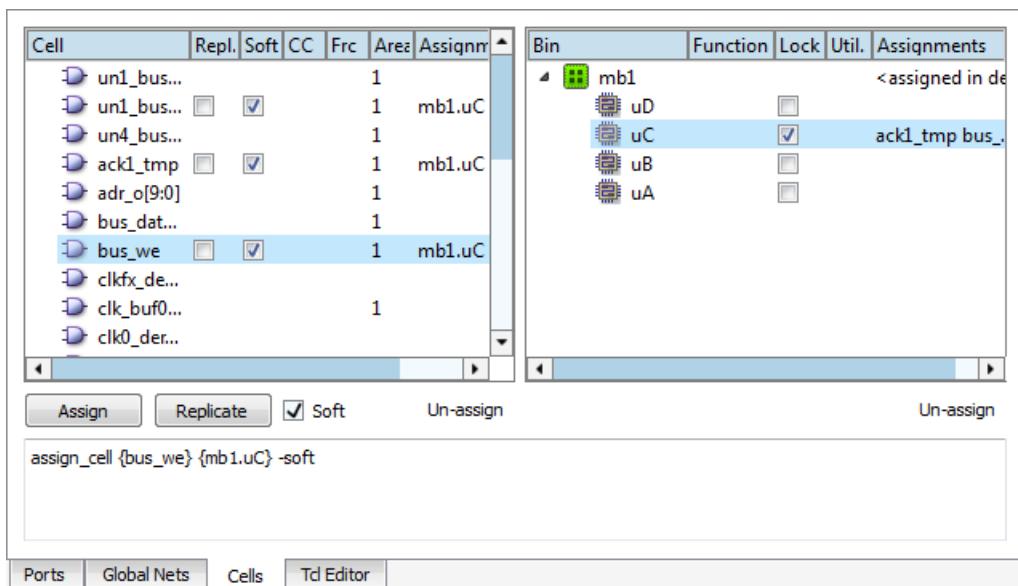
If you want to restrict the associated port cells to only those bins connected to the specified port or external bins, enable the Constrain check box. The tool updates the corresponding CC check box when you click Assign.

- To assign constraints like Soft or Constrain to multiple objects at the same time, use the Ctrl key to select the objects.
- Click Assign. The software displays a T (Trace) or a B (Bin) in the Assignments columns when the assignment is made. Expand the panel as needed to see the Assignments column. The tool also displays the corresponding pcf command line in the panel below the Assign button and in the PCF Editor Tcl pane at the bottom of the window. If the assignment is not legal, the tool displays a message in red instead of the assignment command.



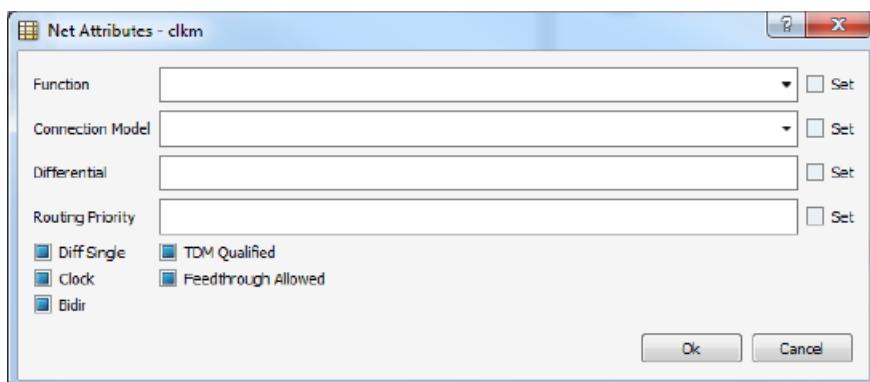
3. To assign system-defined global nets, do the following:

- Select the Global Nets tab. The left pane displays the system-defined global nets.
- To define additional nets for assignment to global traces, enter the net name in the field immediately below the list of nets and click the adjacent plus icon. The net is added to the list of nets. You can also add nets by dragging and dropping from a HDL Analyst view.
- To assign a net, select it in the left pane. Then select the trace to which you want to assign it in the right pane.
- Click Assign. The software displays the corresponding pcf command line in the panel below the Assign button and in the PCF Editor Tcl pane at the bottom of the window. If the assignment is not legal, a message is displayed in red instead of the assignment command.



4. To set attributes for a port or net, do the following:

- Select ports on the Ports tab, or nets on the Global Nets tab. Use this method to set or change the attributes for a set of nets.
- Right-click and select the corresponding Port/Net Attributes command from the popup menu.
- Set the attribute for the net or port. For example you can qualify a net for TDM, or set the connection model.



5. To assign cells to bins, do the following:

- Select the Cells tab. The left pane displays the cells that can be assigned. You can assign them to a bin or replicate them, as described below.
- Select the cell to be assigned in the left pane. In the right pane, select the bin to which you want to assign the cell or where you want it replicated.
- Optionally, specify additional guidance for cell assignment or replication.

To allow the partitioner more freedom to improve cell assignment results, enable Soft before clicking Assign.

For replication you can enable Constrain to restrict the specified cells to only those bins connected to external bins. To force cell replication even when the cell does not include output connections in the target bin, enable Force.

- Specify cell assignment or replication for the selected cells by clicking Assign or Replicate, respectively.

If you set one of the cell assignment or replication options described above, the software enables the corresponding check box when you click Assign or Replicate.

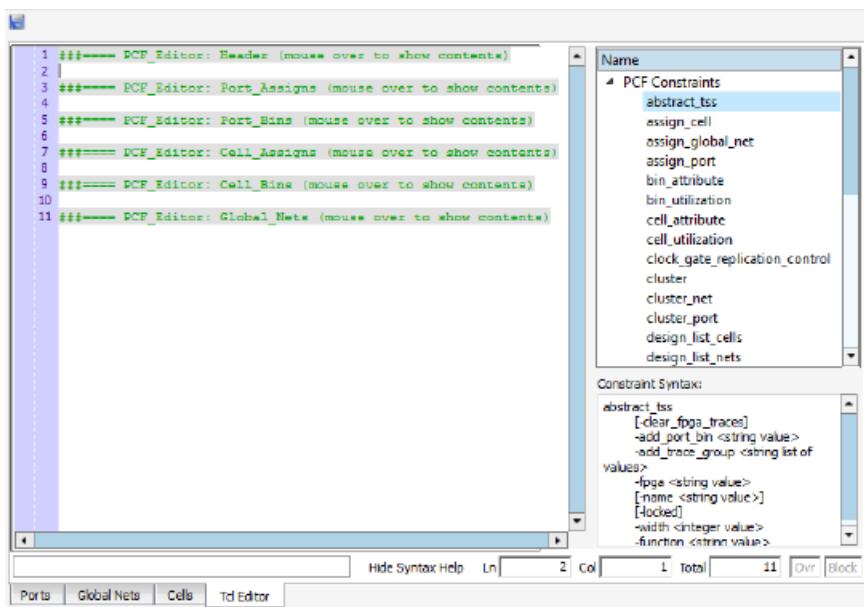
The software displays the corresponding pcf command line in the panel below the Assign and Replicate buttons and in the PCF Editor Tcl pane at the bottom of the window. If the assignment is not legal, the software displays a message in red text, instead of the assignment command.

6. To remove an assignment, select the assigned object in the left pane and click Un-assign.

Clicking Un-assign does not remove the corresponding command from the Tcl window at the bottom.

7. Follow these steps to edit PCF constraints in the GUI editor:

- Open an existing file in the PCF editor and select the Tcl Editor tab. The window shows the file contents, available commands, and the complete command syntax for the selected command.



- To add a PCF command, double-click the command name in the upper right pane. Refer to the Constraint Syntax in the lower right pane to complete the entry in the left pane.

Edit the entry as required. Copy the command from the Tcl window at the bottom and paste it into the file in the left pane, making sure to exclude messages and only copy valid command syntax.

- Close the window when you are done, and save the updated file when you are prompted to do so. There is also a Save icon in the upper left of the window.

For additional details about different kinds of constraints, refer to these sections:

Global setup [Defining Global Information in the PCF, on page 246](#)

Port assignments [Assigning Ports in the PCF, on page 247](#)

Cell assignments [Assigning Cells in the PCF, on page 249](#)

Clock assignments [Assigning Clocks in the PCF, on page 251](#)

Using PCF Queries for TSS and Netlist Information

At different stages of the design you can run pcf queries to get details about design objects or TSS hardware specifications. You can get different information depending on the database state from which you run the query.

1. Start the PCF editor.
 - Make sure you are in one of the partitioning states: pre-partition, partition, or system route.
 - Click on the PCF Editor icon in the GUI or specify this command: `edit pcf pcfFile -mode gui`. You must use the GUI PCF editor to enter the queries. For TSS queries you must open the PCF editor from each state where you make the query.
2. For design queries about objects in the design, use the following commands in the PCF editor:

| To Get ... | PCF Command/Stage |
|---|--|
| A list of nets connected to top-level I/Os | <code>design_list_nets</code> Pre-partition or partition |
| A list of nets connected to top-level I/Os and interconnections | <code>design_list_nets</code> System route, System generate |
| A list of cells | <code>design_list_cells</code> Pre-partition or partition |
| A list of ports | <code>design_list_ports</code> |
| Cell area estimates, resource utilization | <code>design_list_properties -cell</code> Partition |
| Net properties | <code>design_list_properties -net</code> System route |
| Port properties | <code>design_list_properties -port</code> System route |

Many of the commands have arguments that let you further filter the query. For example, you can use `design_list_cells` arguments to filter the list of cells by name, type, parent cell, assignment status, and so on. For the complete command syntax, refer to [Chapter 3, Partition Constraint File Tcl Commands](#) in the *Command Reference Manual*. For examples, see [Design Query Examples, on page 262](#).

If you have multiple overlapping queries, the reported results will be the intersection of the commands.

3. To query the hardware specifications from the tss file, do the following:
 - Open the PCF editor from the partitioning state where you are running the query. For multiple states, you must open the editor in each state.
 - Use these pcf query commands:

| To Get a List of... | PCF Command |
|---|-----------------------|
| TSS pins | tss_list_pins |
| TSS bins | tss_list_bins |
| TSS functions | tss_list_functions |
| Trace names (as defined in the TSS hardware definition) | tss_list_traces |
| TSS trace groups | tss_list_trace_groups |
| TSS properties: trace, bin, pin | tss_list_properties |

For example, you can get a list of trace names before partitioning, using the tss_list_traces command. This command lists all the trace names, according to the hardware setup defined in the TSS.

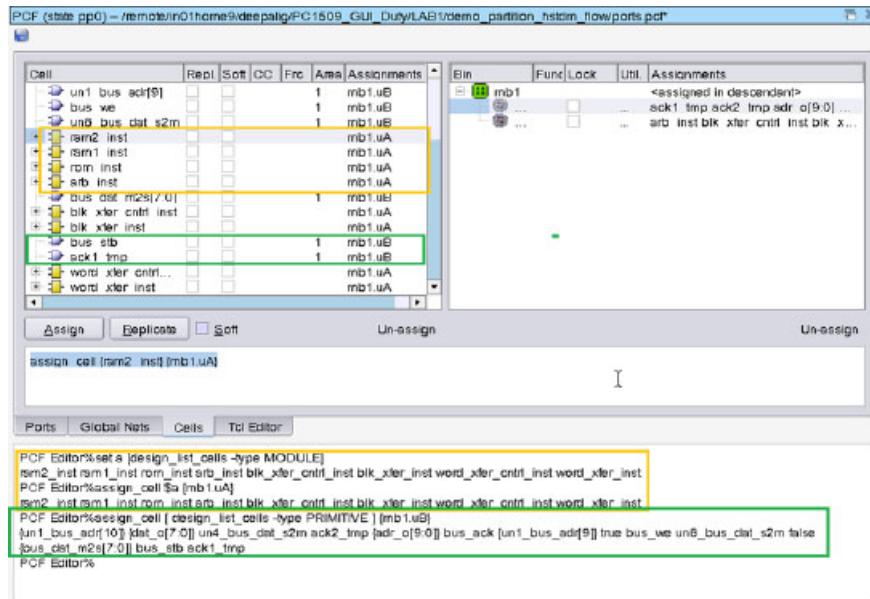
Many of the commands have arguments that let you further filter the query. See [TSS Query Examples, on page 264](#) for some examples, but for the complete command syntax, refer to [Chapter 3, Partition Constraint File Tcl Commands](#) in the [Command Reference Manual](#).

If you have multiple overlapping queries, the reported results will be the intersection of the commands.

Design Query Examples

Example 1: Listing Modules (Pre-Partition)

This example first finds the list of modules in the pre-partition state (design_list_cells -type module) and assigns them to an FPGA bin using the assign_cell pcf command:



Example 2: Listing Port Properties (Partition)

This command is specified from a partition state:

```
design_list_properties -port [adr_o [7]]
```

It lists the properties of the port:

```
TO_PORT [adr_o [9:0]] ASSIGNED 1 BINS TOP_IO_HT3_fb1_B1
```

Example 3: Finding all Clock Nets (System Route)

This command is specified from a system route state:

```
design_list_nets -assigned 1 -clock 1
```

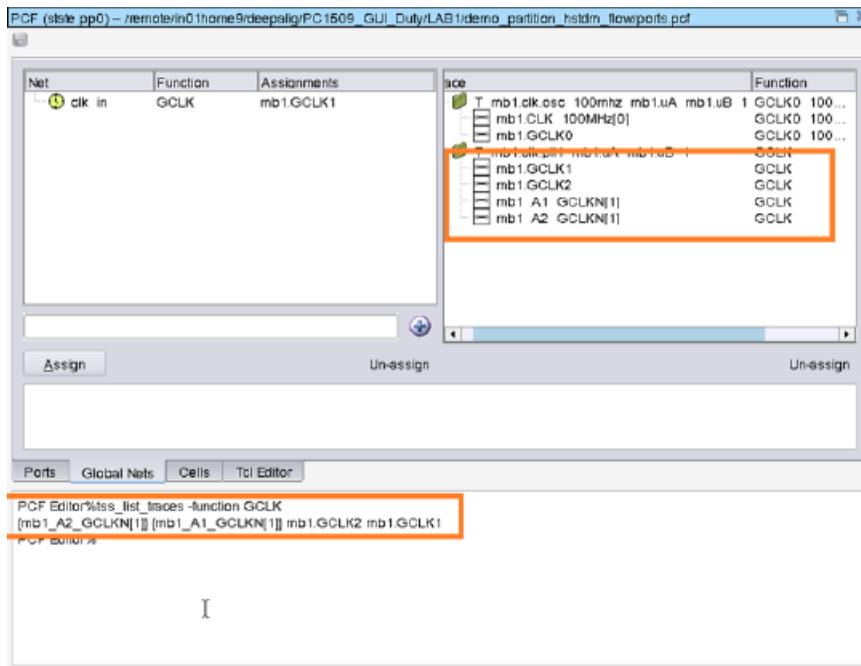
It finds all the assigned nets of type clock:

```
clk_in
```

TSS Query Examples

Example 1: Finding Trace Names (Pre-Partition)

This example shows tss_list_traces run from a pre-partition state to find a list of global clock traces:



Example 2: Finding FPGA Bins and Applying Constraints

This command embeds a tss_list_bins command to find all the FPGA bins, and then applies a constraint to them:

```
bin utilization -bin [tss_list_bins -type FPGA] -resource {DSP 100}
```

Manually Assigning Inter-FPGA Nets to Traces

It is recommended that you use the automatic partitioner for trace assignment. However, if the partitioning results are not ideal, you can manually assign nets to traces. For manual trace assignment, first identify the trace names and then assign the appropriate inter-FPGA nets to them, as described in the following procedure.

1. Define your hardware setup in an initial TSS file.

You must do this to determine the exact trace names, which could differ from the trace names defined in the hardware manuals, and can vary according to the connection setup. Once the setup is defined, the tool can report the correct names, based on the daughter boards defined and how they are connected in your setup.

For example, the setup could have pass-through riser cards, MGBs with PCIe connections, or MGBs with SATA connections, and these differences affect the trace names. If you have more than one card and you give them differentiating names in the TSS, this can affect the trace names as well.

2. Find the trace names using the PCF editor.

- Do an initial partitioning run. This could be the run where you generated the initial automatic partitioning solution.
- Launch the PCF editor from a partitioned state:

```
edit pcf filename
```

- List the available bins by specifying the `tss_list_bins` command in the PCF Editor Tcl window:

```
PCF Editor% tss_list_bins
```

- Specify the `tss_list_traces` command in the PCF Editor Tcl window to find trace names based on the hardware setup defined in the TSS. For example:

```
PCF Editor% tss_list_traces -bins {fb1.uA fb1.uB}
```

The `tss_list_traces` command queries trace information from the TSS file. For more about the command and its syntax, see [tss_list_traces, on page 205](#) in the *Command Reference Manual*.

In this example, the command lists the traces between FPGAs fb1.uA and fb1.uB:

```
{fb1_A3_D[11]} {fb1_A3_D[10]} {fb1_A3_D[9]} {fb1_A3_D[8]}
{fb1_A3_D[7]} {fb1_A3_D[6]} {fb1_A3_D[5]} {fb1_A3_D[4]}
{fb1_A3_D[3]} {fb1_A3_D[2]} {fb1_A3_D[1]} {fb1_A3_D[0]}
{fb1_A3_C[11]} {fb1_A3_C[10]} {fb1_A3_C[9]} {fb1_A3_C[8]}
{fb1_A3_C[7]} {fb1_A3_C[6]} {fb1_A3_C[5]} {fb1_A3_C[4]}
{fb1_A3_C[3]} {fb1_A3_C[2]} {fb1_A3_C[1]} {fb1_A3_C[0]}
{fb1_A3_B[11]} {fb1_A3_B[10]} {fb1_A3_B[9]} {fb1_A3_B[8]}
{fb1_A3_B[7]} {fb1_A3_B[6]} {fb1_A3_B[5]} {fb1_A3_B[4]}
{fb1_A3_B[3]} {fb1_A3_B[2]} {fb1_A3_B[1]} {fb1_A3_B[0]}
{fb1_A3_A[11]} {fb1_A3_A[10]} {fb1_A3_A[9]} {fb1_A3_A[8]}
{fb1_A3_A[7]} {fb1_A3_A[6]} {fb1_A3_A[5]} {fb1_A3_A[4]}
{fb1_A3_A[3]} {fb1_A3_A[2]} {fb1_A3_A[1]} {fb1_A3_A[0]}
{fb1_A2_GCLKN[1]} fb1.GCLK2 {fb1_A3_GCLKN[1]} fb1.GCLK3 fb1.GCLK0
{fb1.CLK_100MHz[0]} {fb1_A1_GCLKN[1]} fb1.GCLK1 fb1.RESETn
```

The next example lists traces between a PCIe and FPGA board:

```
PCF Editor% tss_list_traces -bins {conn3.mgb mb_DXS4.uA}
```

```
{conn3.RXP[3]} {conn3.RXP[2]} {conn3.RXP[1]} {conn3.RXP[0]}
{conn3.RXN[3]} {conn3.RXN[2]} {conn3.RXN[1]} {conn3.RXN[0]}
{conn3.REFCLKP[0]} {conn3.REFCLKN[0]}
{conn3.TXP[3]} {conn3.TXP[2]} {conn3.TXP[1]} {conn3.TXP[0]}
{conn3.TXN[3]} {conn3.TXN[2]} {conn3.TXN[1]} {conn3.TXN[0]}
```

Trace names are also listed in the system route report, if you have run that phase. Check the Trace Assignments section of the system route report.

- After finding the trace names, use them in net assignments, made with the `net_route` PCF command. Define them with the `-using_trace` argument. For example:

```
net_route {a_r[4]} -from {fb1.uA} -to {fb1.uB} -using_trace {fb1_A3_B[4]}
net_route {a_r[3]} -from {fb1.uA} -to {fb1.uB} -using_trace {fb1_A3_B[3]}
net_route {a_r[2]} -from {fb1.uA} -to {fb1.uB} -using_trace {fb1_A3_B[2]}
net_route {a_r[1]} -from {fb1.uA} -to {fb1.uB} -using_trace {fb1_A3_B[1]}
net_route {a_r[0]} -from {fb1.uA} -to {fb1.uB} -using_trace {fb1_A3_B[0]}
```

See [net_route, on page 237](#) in the *Command Reference Manual* for more information about the command syntax.

- For daughter board trace assignments, use `assign_port` commands in the `.pcf` file instead of `net_route`.
- To manually specify that TDM not be used between certain FPGAs, set the `trace_group_attribute` to `DIRECT`.

Use this when you need more control over nets; for example when you do not want to use TDM for low slack nets. DIRECT changes the default assignment, and does not use TDM between the specified FPGAs.

6. Run system route (run system_route) after making the assignments.

Check the actual assignments in the system route report. Commands like the one shown below indicate which traces were assigned.

```
assign_trace a_r[4] fb1_A3_B[4]
```

Working Iteratively with PCF Assignments

The assignment of partitioning constraints is iterative. It is recommended that you use multiple pcf constraint files for different phases and purposes. See [Specifying PCF Constraints for Partitioning, on page 245](#) for a list of suggested pcf files.

Use the method described below to work on constraint assignments for partitioning:

1. Partition the design and analyze the reports generated.

Move logic as permitted by the design, so as to optimize feedthroughs between partitions.

2. Once the feedthroughs have been fixed, assign traces.
3. Assign clocks last.

For HAPS-70 systems, PLL1 drives GCLK 1/7, GCLK 2/8, or GCLK 3/9.

For additional information about clocks, see [Working with HAPS Global Clocks, on page 269](#).

- Check the target system report after partitioning to get the global clock nets, resets, and clock bin names. in the *Reference Manual*
- Use these names when you declare the system clocks and assign them to special bins. For example:

```
# GCLK1
# Clocks driven from PLL1 PLL bin:
net_attribute gclk1_p -function GCLK -differential \
    gclk1_n -is_clock 1
assign_port gclk1_p FB1.clk pll1
assign_port gclk1_n FB1.clk pll1
# Assign trace
assign_global_net gclk1_p FB1.GCLK1

# GCLK5
# Clocks driven from PLL2 PLL bin:
net_attribute clk25_p -function GCLK -differential \
    clk25_n -is_clock 1
assign_port clk25_p FB1.clk pll2
assign_port clk25_n FB1.clk pll2
# Assign trace
assign_global_net clk25_p FB1.GCLK5

# GCLK8
# Clocks driven from FPGA:
assign_port gclk8 FB1.clk bd_clk8
assign_global_net gclk8 FB1.GCLK8
```

Working with HAPS Systems

This section contains some specifics about working with HAPS systems and specifying them in the TSS and PCF files. For comprehensive information refer to the appropriate hardware documentation. Use the following details in conjunction with the information on using the TSS and PCF files ([Defining the System in a TSS File, on page 184](#) and [Defining Constraints in PCF Files, on page 241](#))

- [Working with HAPS Global Clocks, on page 269](#)
- [Defining HAPS-80 Hardware, on page 283](#)
- [Defining a HAPS-DX7 System as a Subsystem, on page 289](#)
- [Defining HAPS-80 Desktop Single/Dual \(HAPS-80D\), on page 291](#)
- [Mapping to Hardware, on page 481](#)
- [Working with HAPS-100 Modules, on page 296](#)

Working with HAPS Global Clocks

The global clock (GCLK) network is distributed to all the FPGAs on a HAPS system. This low-skew, built-in distribution network is used to connect the clock sources with the ports or nets in the user design. HAPS systems have twelve usable global clocks and one fixed-frequency system clock, GCLK0.

| | |
|----------------|---|
| GCLK0 | <p>GCLK0 is a fixed 100 MHz clock and feeds all the FPGAs. Do not use it to drive the user design. It is reserved for HSTDm, UMRBus, CAPIMs, and system IP like GPIO. GCLK0 is also distributed on all HapsTrak® 3 (HT3) connectors.</p> <p>Define GCLK0 when modules interface with the UMRBus. This example defines GCLK0 as driving the GPIO IP.</p> <p>TSS:</p> <pre>board_system_configure -clock FB1.GCLK0 100Mhz</pre> <p>PCF:</p> <pre>net_attribute gpio_clk -function GCLK0_100MHz -is_clock 1 -diffsingle assign_global_net {gpio_clk} FB1.GCLK0</pre> |
| GCLK1 - GCLK12 | <p>GCLK1 - GCLK12 also feed all the FPGAs. Use them to connect the clock source to the user design, through TSS and PCF definitions. The sources for these clocks can be PLL (up to 6 generated clocks), FPGA (up to 6 generated clocks) or external. GCLK1 - GCLK12 do not have a fixed frequency. For details, see xxx and Assigning Internally Generated Clocks to the GCLK Network, on page 272.</p> <p>These clocks are also distributed on HT3 connectors 1-12, with one GCLK per connector.</p> |

On HAPS-80 systems, only FPGA A and B can source global clock nets, as shown in the following table (white cells). The green cells indicate built-in inter-FPGA connections. See [Working with HAPS-80 Resets](#), on page 278 for an example of using a GCLK for reset in a multi-FPGA design.

| FPGA | CLKSRC1 | CLKSRC2 | CLKSRC3 | CLKSRC4 |
|------|-------------------------------|-------------------------------|------------------------------|-------------------------------|
| A | CLK_SRC[1] Drives GCLK1/7 | CLK_SRC[2] Drives GCLK2/8 | CLK_SRC[3] Drives GCLK3/9 | CLK_SRC[4] Drives GCLK4/10 |
| B | CLK_SRC[5] Drives GCLK5/11 | CLK_SRC[6] Drives GCLK6/12 | BC[0] | BD[0] |
| C | CD[0] | CD[1] | CD[2] | BC[0] |
| D | CD[0] | CD[1] | CD[2] | BD[0] |

See these topics for more information:

- [Assigning PLL Clock Sources to the GCLK Network](#), on page 271
- [Assigning Internally Generated Clocks to the GCLK Network](#), on page 272

- [Assigning and Configuring Clocks](#), on page 274
- [Working with GCLK0](#), on page 486
- [Using the HAPS Global Clock Network to Reduce Clock Skew](#), on page 275

Assigning PLL Clock Sources to the GCLK Network

A phase-locked loop (PLL) is a common clock source for design clocks with user-configurable frequencies. HAPS-80 systems provide two PLLs with three programmable outputs each. You can use them to feed the design clocks through the HAPS global clock distribution network (GCLKs).

1. In the tss file, define the clock.

- Set the selected GCLK to source the input from the PLL:

```
board_system_configure -clock fb1.GCLK1 -pll
```

This command identifies a clock source for GCLK1; pll. Do not use GCLK0 to drive a design clock. Consult the documentation for the HAPS system you are using for specifics of the clock distribution network and the exact PLL clock source for the GCLK.

You can override this setting later from Confpro when configuring the board. Some sample choices are clk_src, pll, fpga, clk_left or clk_right.

- Optionally, define the clock frequency for the clock source with the -frequency option:

```
board_system_configure -pll FB1.PLL1_1 -frequency 10000
```

Specifying the frequency in the TSS file allows for a more complete Confpro script to be generated from the TSS file later. Alternatively, you can use cfg_clock_set_frequency command in Confpro to set the frequency when you configure the system (step 5).

2. In the pcf file, connect the clock source to the design clock:

```
net_attribute {designClk1} -function GCLK -diffsingle -is_clock 1  
assign_global_net {designClk1} fb1.GCLK1
```

This command connects the global clock (GCLK1) to the design clock, thus connecting the clock source to the user design.

3. Before running the design, use FDC constraints to declare clocks and their frequencies for synthesis.

4. Check that the constraints and definitions have been properly applied.
 - After pre-partitioning, check the Clock Settings section to make sure that the clocks have been defined correctly.
 - Run report constraint_check to check that there are no errors with the FDC constraints.
 - Check the timing report after system generate and make sure that all the clocks are correctly declared.
5. After running through the entire prototyping flow and generating bit files, use Confpro to configure the frequency.

You can set a total of eight independent frequencies (2 PLLs x 4 outputs). The PLLs can generate frequencies between 0.16 - 350 MHz, 367 - 473.33 MHz, and 550 - 710 MHz. The fourth output can be used for clk_out to clk_in connections.

Assigning Internally Generated Clocks to the GCLK Network

FPGA or internal clocks are clocks that are generated by the user design. Using the global clock network to distribute them across FPGAs ensures that they are phase-aligned at the FPGA boundaries. Do the following to assign an user-generated clock (FPGA clock) to a GCLK.

1. In the tss file, use board_system_configure to configure your selected GCLK to source the input from the FPGA. For example:

```
board_system_configure -clock fb.GCLK2 FPGA
```

This command binds the clock source (FPGA) to GCLK2.

2. Define the functional group of internally generated clocks as GCLKs in the pcf file:

```
net_attribute {clk_gen} -function GCLK -diffsingle -is_clock 1
```

3. Assign the internally generated clock to GCLK in the pcf file:

```
assign_global_net {clk_gen} fb.GCLK2
```

This command connects the global clock (GCLK2) to the internally generated clock nets, thus connecting the clock source to the user design.

4. Run checks after system generate.

- In the Clock Relationships section of the system generate report, check that the user-generated clock or derived clock is reported.
- Check the pin locations to make sure that GCLK is assigned to the internally generated clock.

```
define_attribute {p:clk_gen_0} syn_loc {AB41}
define_attribute {p:clk_gen} syn_loc {AC41}
define_attribute {p:clk_gen} .certify_trace_name {fb.GCLK2}
```

5. Use FDC constraints to declare clocks and their frequencies for synthesis.

This example specifies a clock divider that increases the source clock period by 16x.

```
create_clock -name clk_pll [get_ports clk_pll] -period 10
create_generated_clock -name clk_gen [get_nets {clk_gen}]
-source [get_ports clk_pll] -divide_by 16
```

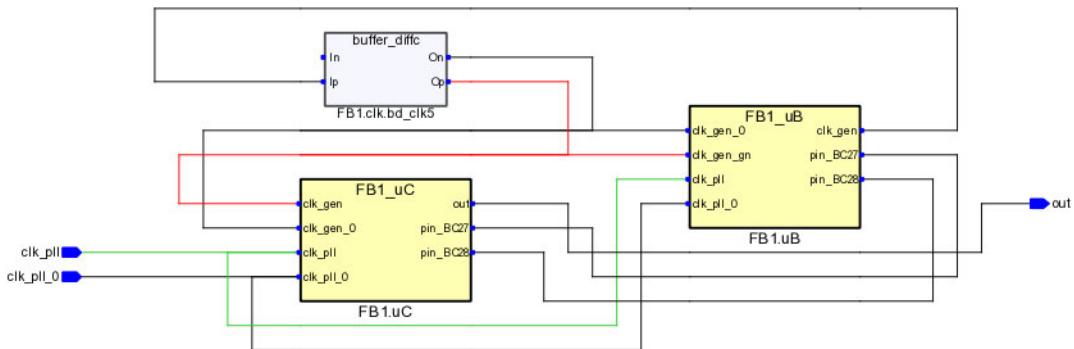
If you set this constraint before system generate, the report shows the clock as a generated clock (clk_gen) instead of a concatenated name.

6. To distribute a generated clock to other FPGAs, use the GCLK network.
 - Define the generated clock using FDC constraints.
 - Check the hardware manuals to find out which GCLKs you can use to drive the clocks from a particular FPGA. For example, to drive a clock from FPGA B to FPGA C, you can use GCLK5, GCLK11, GCLK56, or GCLK12. Set up the TSS configuration accordingly. This statement declares GCLK5 on FPGA B as an FPGA-sourced clock:

```
board_system_configure -clock FB1.GCLK5 fpga
```

- Connect the generated clock to the GCLK you just defined, using PCF constraints. The generated clock now goes from FPGA B through GCLK5 and the global clock network to FPGA C.

```
net_attribute clk_gen -function GCLK
assign_global_net {clk_gen} {FB1.GCLK5}
```



Assigning and Configuring Clocks

1. Identify trace names for assignment, by following these steps:
 - After an initial partitioning run, specify the `report target_system -tss fileName` command to list board bins and traces.
 - Export the report.
 - Open the report and use these trace names for assignment. See [Checking the Target Specification \(TSS\), on page 358](#) for more information about using this report.
2. To assign specific clocks to a trace, use PCF `assign_global_net` constraints, as shown in the excerpt shown below:
 - The PCF `assign_global_net` command must be used directly with a trace name, which you find using the procedure described in step 1. For example:


```
net_attribute {GCLK3_P} -function GCLK -differential
              GCLK3_N -is_clock 1
          assign_global_net {GCLK3_P} {fb1.GCLK3}
```
 - Configure the clock with a `board_system_configure -clock` command in the TSS file.
3. If you do not care about specific clock assignments, follow these steps to let the tool make the assignments:
 - Use `assign_port` syntax like this in the PCF file:

```

net_attribute {GCLK3_P} -function GCLK -differential
    GCLK3_N -is_clock 1
assign_port GCLK3_P fb1.GCLK3

```

- Configure the clock with a `board_system_configure -clock` command in the TSS file.
4. To handle differential clocks, follow these instructions.
- Use pad cells like IBUFDS, IBUFGDS, or OBUFDS for differential signals, because differential signals are on the pads and outside the FPGA, not inside the FPGA. Everything inside the FPGA is single-ended. Differential pad cells convert the differential signals to single-ended signals inside the FPGA.
- Although you can use MMCMs with differential or single-ended clocks outside the FPGA, remember that the MMCMs are internal to the FPGA and do not know about differential signals outside the FPGA.
- If you have one clock feeding a differential buffer, use `assign_global_net` syntax for the clock:

```

assign_global_net clkname_p globalClkTrace
net_attribute -clkname -function {GCLK} -diffsingle clkname_n -is_clock 1

```

- For two clocks feeding a differential buffer, use `assign_port` syntax:
- ```

assign_port clkname_p clockbin
assign_port clkname_n clockbin
net_attribute -clkname -function {GCLK} -differential
 clkname_n -is_clock 1

```

5. If the design uses GCLK0, make sure to follow the recommendations described in [Working with GCLK0, on page 486](#).

## Using the HAPS Global Clock Network to Reduce Clock Skew

Partitioning can cause clock nets to cross FPGAs; the crossing clocks are also called cut clocks. These clocks cause clock skew, which can cause the design to fail on the board.

Do the following to avoid cut clocks:

1. Connect the internal clock net to the HAPS global clock network using a pcf constraint.

You do not need to edit the RTL. The following example assigns the clock net to GCLK7.

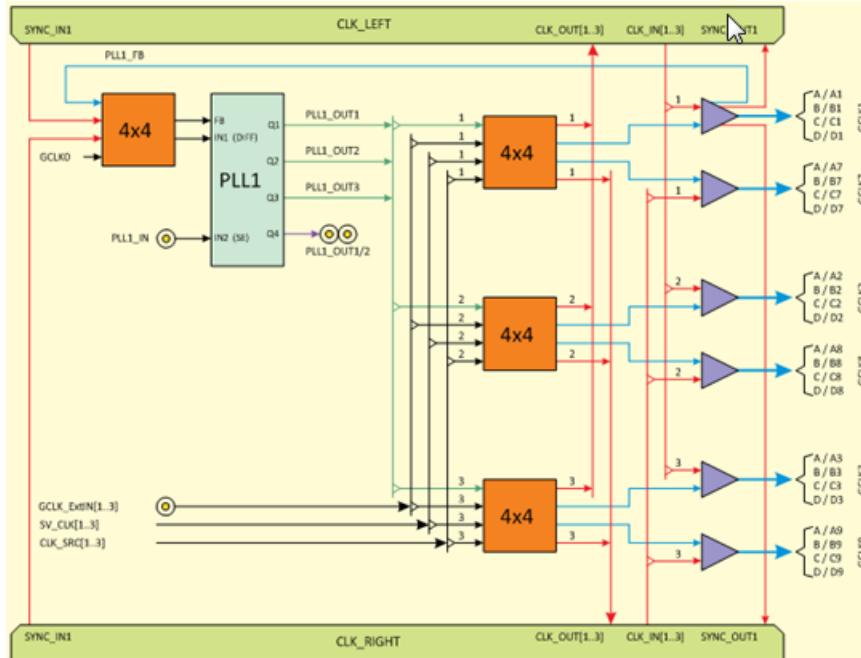
```
net_attribute clk_out -function GCLK -diffsingle
assign_global_net clk_out fb1.GCLK7
```

2. Connect the global clock back to all the FPGAs.

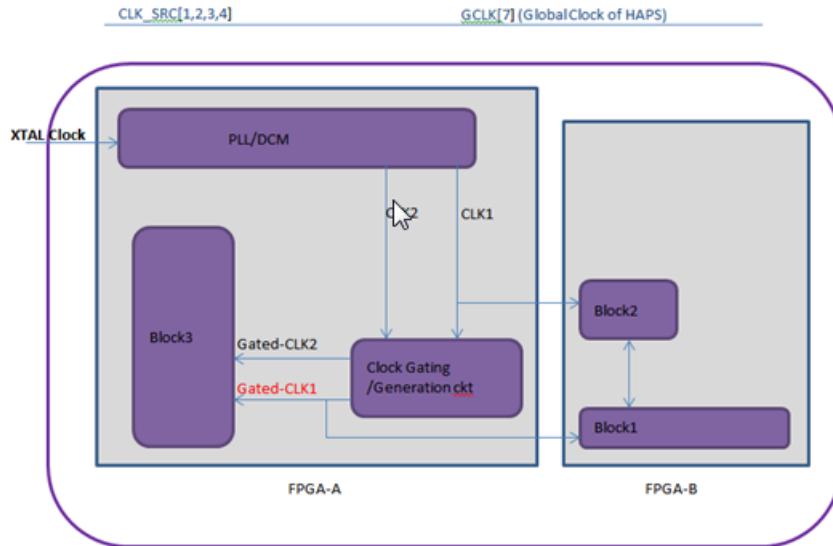
Using the global clock network prevents the clock nets from crossing FPGA boundaries. The tool also replicates the clock-gating circuit on all the FPGAs.

## Example

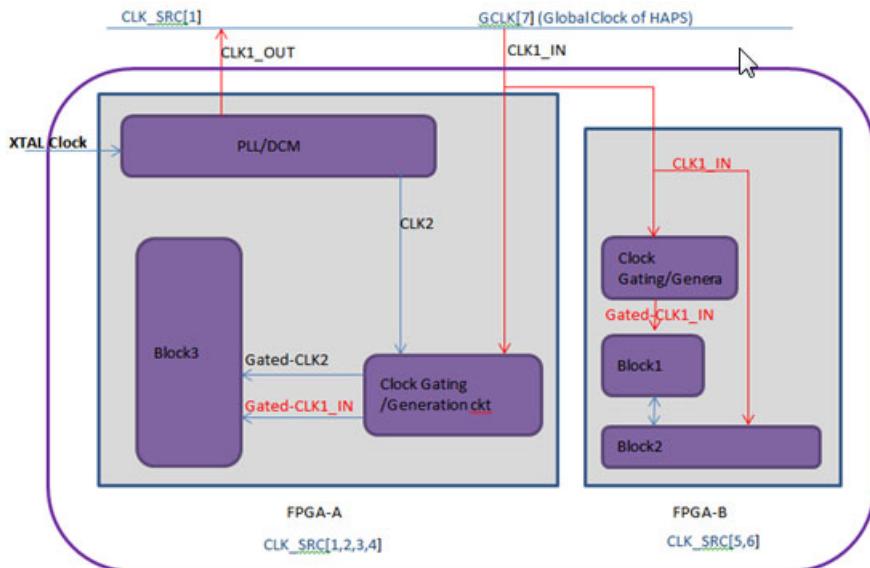
The following example shows a clocking scheme with a single PLL. Any of the GCLKs can be used in the FPGA design. GCLK0 is fixed at 100 MHz and the rest are PLL-generated clocks with configurable frequencies. The FPGA connected to the daughter board inputs the clock and redistribute it to the other FPGAs through a global clock net.



After partitioning, the design has two cut clocks which cross the FPGA boundary: CLK1 and Gated-CLK1.



To avoid cut clocks in this case, connect **CLK1** to **CLK\_SRC[1]** of **FPGA A**. Then connect it back to all the FPGAs using a pcf constraint to assign it to the global clock **GCLK[7]**. The example shows this scenario, with the **clk\_out** clock generated from the **CLOCK\_GENERATION** block connected to the HAPS global clock network.



## Assigning and Distributing HAPS-80 Reset

The HAPS-80 system does not have a global reset, unlike the `RESETn` on previous HAPS systems. Instead each FPGA has a local reset that is part of the System IP (described in [HAPS-80 Architecture Overview, on page 283](#)) and is controlled externally through Confpro. This means that the global reset may not be synchronous across all the FPGAs, and you must handle global reset distribution.

For more information, see these topics:

- [HAPS-80 Reset Configurations](#), on page 278
- [Working with HAPS-80 Resets](#), on page 278
- [Synchronizing Resets](#), on page 280

### HAPS-80 Reset Configurations

There are three ways to configure HAPS-80 resets for multi-design mode:

- Reset set from HAPS System IP

With this method, the Confpro and the supervisor uses the HAPS-80 System IP to control the reset externally, with one FPGA acting as the master. The output reset from the master System IP is the global reset source and the local FPGA resets are synchronized to GCLK0 (100 MHz).

- Reset set from an HT3 connector

This method bypasses the System IP and uses a GPIO daughter board or top-level pin to externally control the reset from a controller or button. The local FPGA resets (being controlled through top-level pins) are asynchronous unless they are explicitly synchronized to a user clock region.

- Reset set as a combination of System IP and user logic

This method uses both the System IP as well an HT3 pin, MMCM, or other user logic. Combined local FPGA resets should be considered asynchronous unless they are explicitly synchronized to a user clock region.

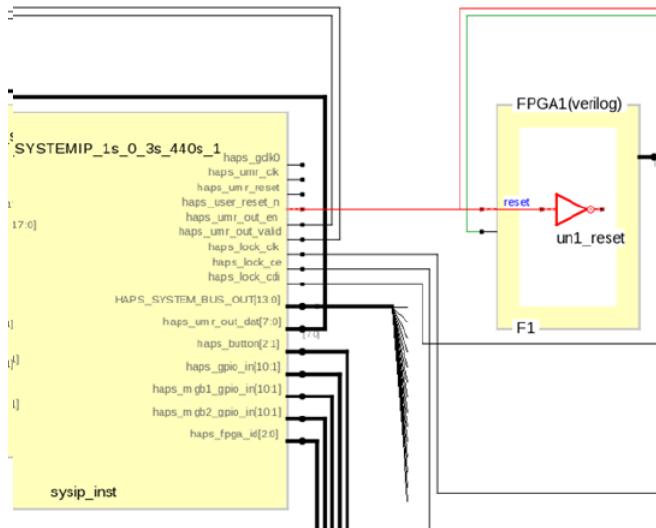
### Working with HAPS-80 Resets

On HAPS-80 systems, each FPGA has a local reset that is part of the System IP. The following procedure provides the basics of specifying the resets on HAPS-80 systems.

For detailed information about HAPS-80 resets, refer to the *Synchronous Reset in Multiple FPGAs with ProtoCompiler* application note, which is available under the *HAPS-80 Series* heading on HAPS SupportNet.

1. Assign the reset, using the local reset that is part of the System IP using pcf commands as shown. The example below assumes there is a signal called *reset*.

```
net_attribute {reset} -function SINGLE_FPGA_RESET
assign_port {reset} -trace {FB1.A_USER_RESETN}
```



2. Specify how the reset is to be synchronized, using the *reset\_synchronize* command in the pcf file.

```
reset_synchronize -toplevel_net {reset} -clock {mmcm_clk}
 -init {0} -extra_pipeline_stages {0}
```

Follow the guidelines in [Synchronizing Resets, on page 280](#).

As there is no global reset, the reset must be synchronized and distributed across the FPGAs. Like clock trees, the reset signal must be distributed and synchronized to eliminate erratic design behavior.

## Synchronizing Resets

You must synchronize resets because there is no global reset. To ensure that behavior is predictable and timing results are accurate, resets must be synchronized and distributed across the FPGAs.

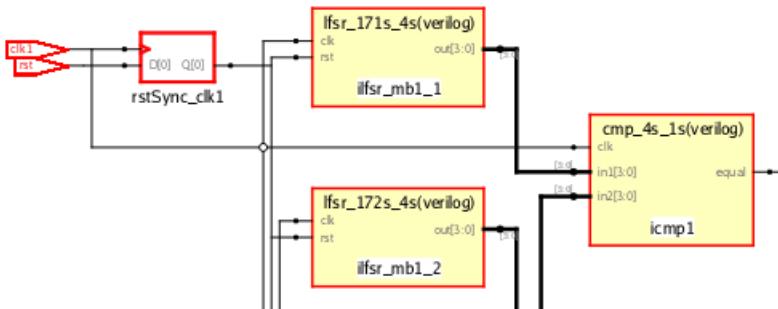
To do this, use the PCF `reset_synchronize` command. This example shows active low reset synchronization:

```
reset_synchronize -toplevel_net{rstn} -clock {clk} -init {0}
```

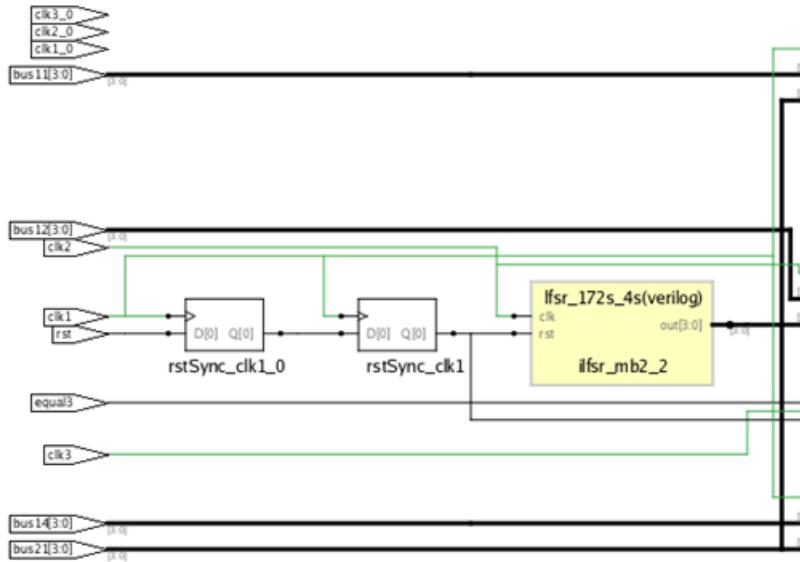
For complete syntax details, refer to [reset\\_synchronize, on page 194](#) in the *Command Reference Manual*. You can also use this command to synchronize other asynchronous signals. For more about reset synchronization refer to this article from the HAPS Prototyping newsletter: *Automating Synchronous Signal Distribution in Multiple FPGAs with HAPS ProtoCompiler*.

When defined properly, the reset signals are synchronized across the design, while maintaining original reset behavior. There is a reset propagation delay.

The command takes effect during the partition and system route phases and automatically creates tree structures for reset synchronization and distribution. It inserts synchronizer elements as specified in the pcf as well as additional elements to ensure that all modules are synchronized across the same level and globally. The following figure shows a design before partitioning:



The next figure shows the design after system route:



The following are some guidelines to using the `reset_synchronize` command:

1. Define the reset net.

If the RTL has a single reset, the partitioned design must also have just one reset, which must be synchronized and distributed across the partitions, using the `reset_synchronize` command.

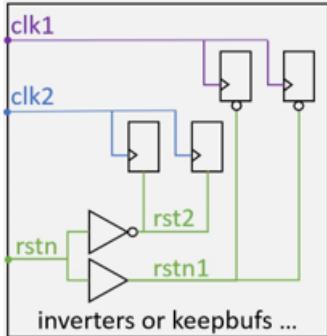
- Define the reset net on a top-level FPGA port or an internal net at the top level of the design. Use the `-toplevel_ff` argument to specify a top-level flip-flop to be used for reset synchronization. The command replicates this reset synchronizer flip-flop on all FPGAs that are a load on the global reset. Alternatively, you can define a top-level net with `-toplevel_net`.
- Do not define a net with combinatorial loads like IBUFG or logic inverters as the reset net. Instead, specify the net directly connected to the flip-flop reset ports as the reset net.
- Specify the type of reset initialization with the `-init` argument to the `reset_synchronize` command. Use `-init {0}` for active low and `-init {1}` for active high.

2. Define a unique reset net for each separate clock domain.

If the design has multiple reset signals, each signal must be treated individually and synchronized and distributed.

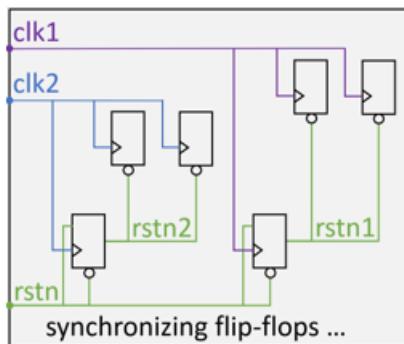
- Depending on the design, use syn\_keep, inverters or buffers to separate the nets for different domains. Then define the resets with reset\_synchronize. For example:

```
reset_synchronize -toplevel_net {rstn1} -clock {clk1} -init {0}
reset_synchronize -toplevel_net {rst2} -clock {clk2} -init {1}
```



- To synchronize the same reset in multiple clock domains, duplicate the net. You can use existing synchronous nets or use flip-flops to duplicate the nets. For example:

```
reset_synchronize -toplevel_net {rstn1} -clock {clk1} -init {0}
reset_synchronize -toplevel_net {rstn2} -clock {clk2} -init {0}
```



3. Define the clock to be used for reset distribution at the top level of the design.

- Use the `-clock` argument to the `reset_synchronize` command to specify the clock to synchronize the registers.
  - If you specify MMCM clocks for synchronization use caution when replicating them, as they can affect clock skew and phase. The reset distribution pipeline must wait for MMCMs to lock before release.
4. Check results.
- After partitioning, check the report to see what was replicated. Replication indicates loads on the nets. Guide automatic replication with the `-force_repl` and `-repl_bins` arguments to `reset_synchronize`.
  - After system route, check the report for the number of pipeline stages and flip-flops added to the design for synchronization.

## Defining HAPS-80 Hardware

HAPS-80 systems, which are available in single-FPGA, two-FPGA, and four-FPGA versions, have some hardware architecture innovations that must be correctly handled in the TSS and PCF files.

This section is only an introduction; for details about the hardware architecture, refer to the HAPS-80 documentation on HAPS SupportNet on SolvNet.

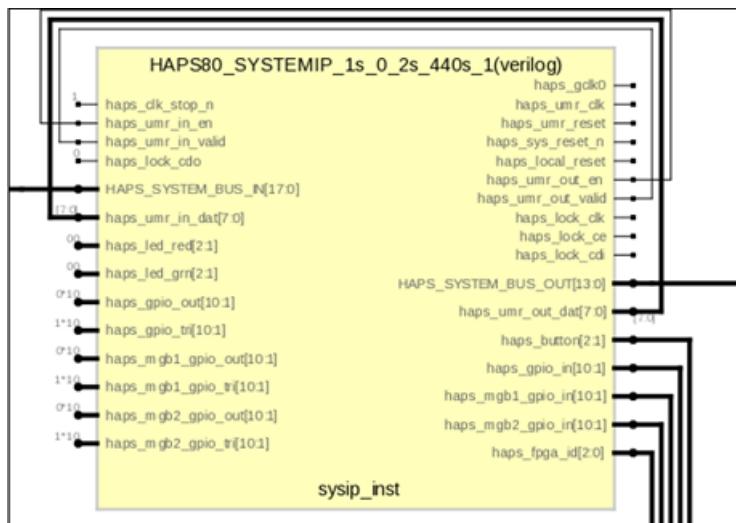
- [HAPS-80 Architecture Overview](#), on page 283
- [Guidelines for HAPS-80 Interconnect](#), on page 285
- [Assigning and Distributing HAPS-80 Reset](#), on page 278
- [Working with HAPS-80 GPIO](#), on page 286

## HAPS-80 Architecture Overview

For complete details of the HAPS-80 hardware, see the hardware documentation. The following information covers some features as they affect prototyping with the ProtoCompiler software.

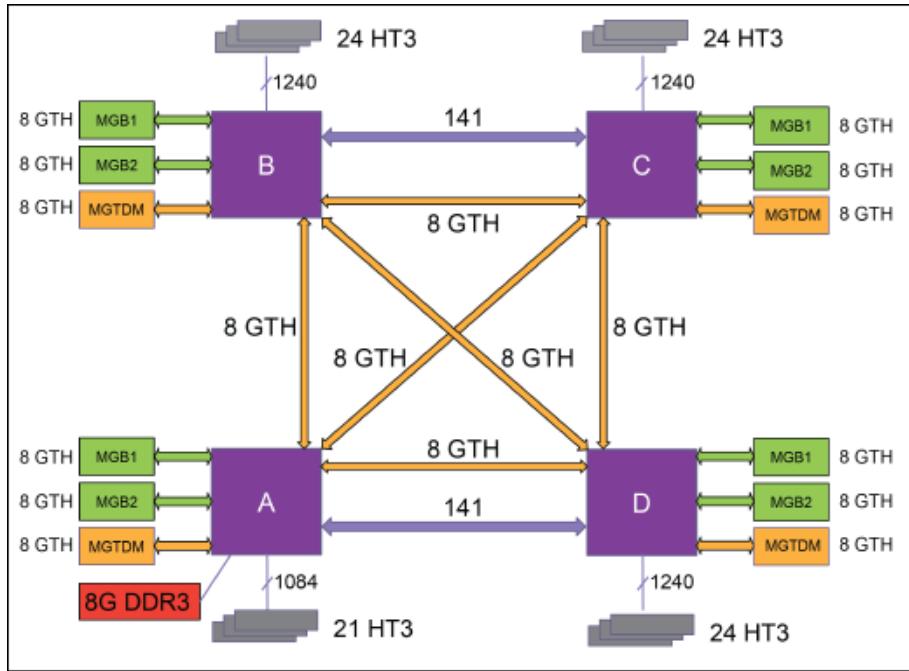
### HAPS-80 System IP

The System IP is a module that the ProtoCompiler automatically inserts for each HAPS-80 FPGA. It consolidates various operations including multiplexing, serialization and de-serialization, local resets, GCLK0 de-skewing, UMR chaining, and trace assignment for GPIO, buttons, and LEDs.



## HAPS-80 Inter-FPGA Traces and Built-in Memory

The HAPS-80 system has dedicated inter-FPGA traces between FPGAs A and D and C as shown below for the 4-FPGA HAPS-80 S104. The two-FPGA version is similar, with inter-FPGA traces between A and B. There are 141 onboard traces between the FPGA pairs.



In addition, FPGA A on all HAPS-80 versions (4-FPGA, 2-FPGA, and single FPGA) includes 8 G of DDR storage that can be used for debug.

## Guidelines for HAPS-80 Interconnect

There are 24 HT3 connectors per HAPS-80 FPGA module, and 2 MGB connectors per FPGA module for MGB cards and MGB riser cards. The system uses the HapsTrak 3 connector standard, and supports 52 signals per connector.

Use the following guidelines to connect HAPS-80 systems:

- Remove A1, A2, and A3 from the TSS definition. They are reserved for debug and cannot be used as top I/Os or HT3 connections.
- You cannot configure the voltage for the 144 fixed/buried traces. The voltage is fixed at 1.8 V.
- J12 is the HR bank and cannot be used for HSTDMD.
- J16-19 are limited pin connectors.

Refer to the HAPS-80 documentation for more information about the architecture.

# Working with HAPS-80 GPIO

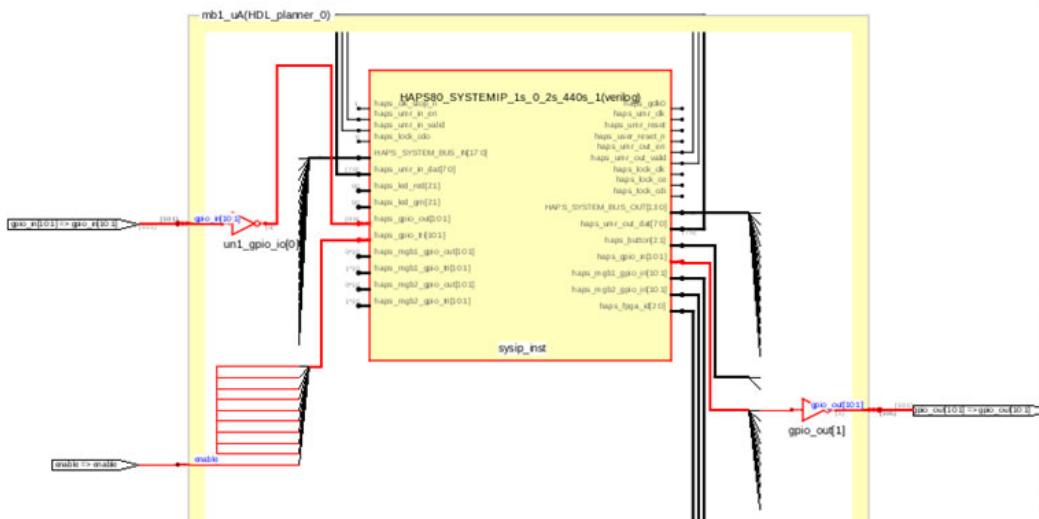
The consolidated IP includes virtual pins that you can use directly for GPIO, instead of the separate GPIOLINK IP used with the HAPS-70 and HAPS-DX systems. During partitioning, these pins appear as FPGA pins and can be used directly. The tool then moves the connections from virtual FPGA pins to System IP pins. It automatically adds the System IP module, makes all the connections needed, and adds pin location constraints, I/O standards, and so on.

1. Make RTL changes to create the tristate signal.

The System IP handles tristates, and has one signal instead of `_in`, `_out`, and `_tri`. You might have to add bidirectional buffers to support tristate signals, using the `define_haps_io` command:

```
define_haps_io {p:MGB4_GPIO[10:1]}\n-haps io {AM1 GPIO[10],AM1 GPIO[9], ...}
```

The output of the tristate buffer is connected to the `haps_gpio_out` pin on the System IP. The input is connected to `haps_gpio_in` and the enable is connected to `haps_gpio_tri`. The buffer logic is handled inside the System IP.



2. To find the virtual System IP pins, refer to the Virtual Pins section at the bottom of the `haps_io_report.txt` file.

This file is generated after the premap stage. Use the virtual pins directly for GPIO. The Virtual Pins section also lists the LED pins, `A_LED_RED` and `A_LED_GRN`.

3. Do not assign pin locations for GPIO in and out signals, as the system takes care of this automatically.

The `GPIO_1_tri`, `GPIO_1_out`, and `GPIO_1_in` signals are uni-directional. The System IP handles them as a single bi-directional GPIO signal.

4. In unpartitioned single-FPGA designs, do not use `syn_loc` on the virtual GPIO pins, but use `define_haps_io` instead to specify pin locations.

`define_haps_io` is HAPS-aware and supports these virtual MUXed pins.

5. For partitioned designs, assign the virtual pins to traces.

- To specify a particular trace to use, use syntax like the example below:

```
net_attribute {constant_compare} -function USER_LED
assign_port {constant_compare} -trace {{fb1.B_LED_GRN[2]}}
```

This example assigns a trace from the `usr_b_led` bin:

```
net_attribute {constant_compare} -function USER_LED
assign_port {constant_compare} {fb1.usr_b_led}
```

This example selects a GPIO trace from a virtual bin that connects the `GPIO_*`, `AM*_*`, `BM*_*` traces, based on functional group and bin connectivity:

```
net_attribute {constant_compare} -function USER_GPIO
assign_port {constant_compare} {virtual_port}
```

- Use matching names for the virtual pin in the board file and the IP bit ports.

6. Connect slow peripherals like the LEDs and buttons.

HAPS-80 systems include 10 panel GPIOs, 20 MGB GPIOs, 2 buttons, and 4 LEDs per FPGA. You can assign top-level ports to System IP traces, keeping them in functional groups. The System IP MUXes/deMUXes and connects the traces to the supervisor FPGA, which in turn MUXes/deMUXes the traces and connects them to the

physical LEDs and buttons on the HAPS system. The maximum speed is 400 KHz.

- To assign traces for panel GPIO:

```
net_attribute {data[9:0]} -function SLOW_PANEL_GPIO
assign_port {data[9:0]} {virtual_port}
assign_port {data[9:0]} -trace {fb1_A_GPIO[10:1]}
```

- To assign traces for MGB GPIO:

```
net_attribute {out[9:0]} -function SLOW_MGB_GPIO
assign_port {out[9:0]} {virtual_port}
assign_port {out[9:0]} -trace {fb1_AM1_IO[10:1]}
```

- To assign traces for a button:

```
net_attribute {data[1:0]} -function PANEL_BUTTON
assign_port {data[1:0]} -trace {fb1.B_BUTTON[2:1]}
```

- To assign traces for LEDs:

```
net_attribute {out[3:0]} -function PANEL_LED
assign_port {out[1:0]} -trace {fb1.A_LED_GRN[2:1]}
assign_port {out[3:2]} -trace {fb1.A_LED_RED[2:1]}
```

7. To hook up the GPIOLink MGB sideband signals for the HAPS-80 system, do the following:

- Refer to the manual for the MGB interface daughter board for GPIO pin information. For example, the PCIE reset on the MGB card is J1 (pin 35).
- Map it accordingly: MGB card J1(pin 35) -> pin on Riser (J2) -> pin on MGB slot (IO[1]).

Use `define_haps_io` to map it to the System IP pin. The PCIE reset maps to AM1\_IO[1] or AM2\_IO[1]. The bi-directional MGB signals on System IP are AM1\_IO and AM2\_IO. They are the equivalent of the tri/in/out signals on the HAPS-70 MGB1\_GPIO.

You might also have to add bidirectional buffers to support the tristate signal, using the `define_haps_io` command:

```
define_haps_io {p:MGB4_GPIO[10:1]}
-haps_io {AM1_GPIO[10],AM1_GPIO[9], ...}
```

## Defining a HAPS-DX7 System as a Subsystem

You can chain a HAPS-DX7 board system (HAPSDX7\_S4 or HAPSDX7\_S6) and use it as a subsystem, or for debug and validation.

The HAPS-DX7 has ten HapsTrak 3, four MGB, and two HSIO connectors. It has four global clocks distributed to the FPGA. Clock GCLK0 is fixed at 100MHz for system functions, and is synchronized between multiple systems to guarantee the consistent transfer of data. Externally generated clocks can be connected to HAPS-DX7 using coax connectors. PLL\_IN provides a reference input to the PLL. PLL\_OUT1 and PLL\_OUT2 are single-ended or differential signaling clock outputs from the PLL and have the same frequency. There are two SMB coax clocks directly connected to the user FPGA.

The FPGA controls four bi-colored LEDs and 66 GPIO signals which are distributed as shown in the following table. The GPIO signals in the MGB connector and HSIO connector are controlled via the GPIOLink IP.

| Number of GPIO Signals | Connectors         |
|------------------------|--------------------|
| 40                     | MGB connectors     |
| 10                     | 2 mm 14-pin header |
| 16                     | HSIO connectors    |

There are no CLK\_LEFT and CLK\_RIGHT connectors.

Follow these steps to connect a HAPS-DX7 system to a HAPS-70 system:

1. Instantiate the HAPS-DX7 board system in the TSS file.
  - You must include a HAPS-70 module in the chain. You cannot have just HAPS-DX7 systems in the chain.
  - Specify the HAPS-DX7 system.
  - Connect the ten HT3 connectors on the HAPS-DX7 system with the HT3 connectors on the HAPS-70 system. Connect the HAPS-70 CDE\_OUT to the HAPS-DX7 CDE\_IN. Set the HAPS-DX7 GCLK0 to cde\_in.
  - If you have an FMC (FPGA Mezzanine Card) board connected to the HAPS-DX7 system, connect the FMC\_ADAPTER\_HT3 board to the four HT3 connectors and one HSIO connector on the HAPS-DX7 system. Assign the traces connected to the daughter board.

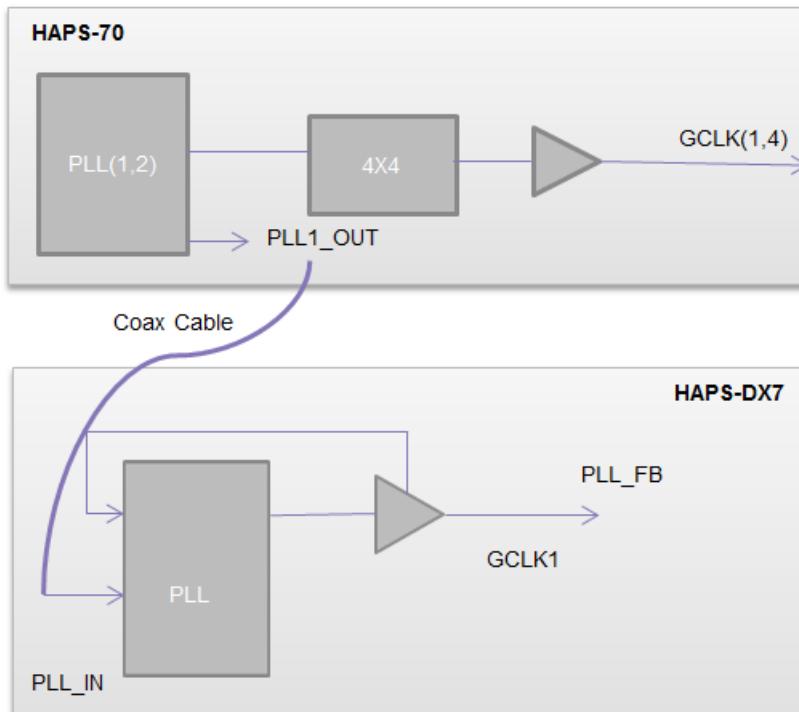
2. Follow these partitioning guidelines:

- Pre-assign IP to the HAPS-DX7 board system using pcf constraints.
- Set other constraints to restrict logic assignment to the HAPS-DX7 system by the auto-partitioner.
- Use tdm\_control -type HSTDMD for inter-FPGA data nets.

3. Synchronize the HAPS-DX7 clocks with the HAPS-70 clocks.

HAPS-DX7 systems do not have CLK\_LEFT and CLK\_RIGHT connectors, so the clocks must be synchronized.

- To synchronize the PLLs, use specific-length coax cables to connect PLL1\_OUT(1,2) on the HAPS-70 system to PLL\_IN on the HAPS-DX7 system.
- Configure GCLK1 on the HAPS-DX7 system to be the same frequency as GCLK1 and PLL1\_OUT(1,2) on the HAPS-70 system.



## Limitations

Currently there are some limitations to incorporating a HAPS-DX7 system:

- No TSS support for GCLK synchronization through PLL\_OUT<->PLL\_IN connection.
- HSTDIM is the only mode allowed for TDM, because GCLK traces are not available across boards. GCLK0 is available through the CDE chain.
- GPIO header does not permit the assignment of traces to 10 GPIO pins on the board.

## Defining HAPS-80 Desktop Single/Dual (HAPS-80D)

Technology: HAPS-80D

Board Names: HAPS80D\_DUAL and HAPS80D\_SINGLE

---

**Note:** HAPS-80D does not support synthesis flow. Hence **haps\_define\_io** command is not supported on HAPS-80D. The device supports only partition flow.

---

Example TSS command for adding a HAPS-80D Dual.

```
board_system_create -add HAPS80D_DUAL -name fb1 -speed_grade {-1-c}
```

- I/O Connectors
- Clocking
- Synchronizing Clocks Across Two HAPS-80D

### I/O Connectors

HAPS-80 Desktop includes connectors for software debugging and user designs. For more information about the connectors, see *HAPS-80 Desktop Reference Manual*.

| Connector    | TSS name    |
|--------------|-------------|
| MICTOR38     | mictor38    |
| CoreSight 20 | coresight20 |
| PMOD         | pmod        |

| Connector | TSS name |
|-----------|----------|
| JTAG20    | jtag20   |
| GPIO1     | gpio1    |
| GPIO 2    | gpio2    |

Some connectors share FPGA A pins and cannot be used at the same time. The following table lists the invalid combinations.

|                          |
|--------------------------|
| CORESIGHT 20 - JTAG 20   |
| MICTOR 38 - CORESIGHT 20 |
| MICTOR 38 - GPIO 2       |
| MICTOR 38 - JTAG 20      |
| PMOD - GPIO 1            |

This TSS example sets the connectors GPIO2 and PMOD on the first HAPS-80D and the CORESIGHT20 connector on the second HAPS-80D to be active. See [board\\_system\\_configure, on page 245](#) in the Command Reference.

```
board_system_configure -function {fb1 gpio2 fb1.pmod
fb2 coresight20} true
```

## Clocking

Each FPGA has 12 differential design clocks GCLK1-GCLK12. GCLK1-GCLK6 are sourced from the PLLs or FPGAs. GCLK7-GCLK12 are sourced from the PLLs or the clock connector CLK\_IF.

PLL1 sources GCLK1 - GCLK3 and GCLK10. PLL2 sources GCLK4-6 and GCLK11. PLL3 sources GCLK 7-9 and GCLK12. CLK\_IF is used to source clocks across two HAPS-80D using the special clock cable, CLK\_CABLE\_80D.

Example TSS commands for connecting two HAPS-80D.

```
board_system_configure -clock fb1.GCLK1 fpga
board_system_configure -clock fb1.GCLK2 fpga
board_system_configure -clock fb1.GCLK3 fpga
board_system_configure -clock fb1.GCLK5 pll
board_system_configure -clock fb1.GCLK6 pll
board_system_configure -clock fb1.GCLK4 pll
```

```
board_system_configure -clock fb2.GCLK10 clk_if
board_system_configure -clock fb1.CLK_IF pll
board_system_configure -clock fb2.GCLK1 fpga
board_system_configure -clock fb2.GCLK2 fpga
board_system_configure -clock fb2.GCLK3 fpga
board_system_configure -clock fb2.GCLK5 pll
board_system_configure -clock fb2.GCLK6

board_system_configure -connect {fb1.CLK_IF fb2.CLK_IF}
```

## Synchronizing Clocks Across Two HAPS-80D

Two HAPS-80D systems can be chained for up to 4 FPGA-system. Note that HAPS-80D cannot be chained with HAPS-80 or HAPS-70 systems.

A TSS example to synchronize clocks across two HAPS-80D systems. This example uses GCLK1 from the first HAPS-80D. The clock connector on the HAPS-80D is CLK\_IF.

```
board_system_configure -clock fb1.GCLK1 pll
board_system_configure -clock fb2.GCLK7 clk_if
board_system_configure -connect {fb1.CLK_IF fb2.CLK_IF}
board_system_configure -clock fb1.CLK_IF pll
```

## HAPS-80D TSS File Examples

### Example 1 - Connecting Two HAPS-80D

```
board_system_create -haps -name mysystem
board_system_create -add HAPS80D_DUAL -name fb1 -speed_grade {-1-c}
board_system_create -add HAPS80D_DUAL -name fb2 -speed_grade {-1-c}
board_system_configure-top_io{fb1.A8}

board_system_create -interconnect -manual CON_CABLE_100_HT3 -
 name conn6
 -connector {fb1.A5 fb2.B6}
board_system_create -interconnect -manual CON_CABLE_100_HT3 -
 name conn5
 -connector {fb1.A6 fb2.B7}
board_system_create -interconnect -manual CON_CABLE_100_HT3 -
 name conn1
 -connector {fb1.A7 fb2.B8}
board_system_create -interconnect -manual CON_CABLE_100_HT3 -
 name conn2
 -connector {fb1.A9 fb2.B10}
board_system_create -interconnect -manual CON_CABLE_100_HT3 -
```

```

name conn3
-connector {fb1.A10 fb2.B11}
board_system_create -interconnect -manual CON_CABLE_100_HT3 -
name conn4
-connector {fb1.A11 fb2.B12}

board_system_configure -voltage { fb1.A4} 1.2
board_system_configure -clock {fb1.GCLK1} pll
board_system_configure-clock{fb2.GCLK7}clk_if
board_system_configure-clock{fb1.GCLK7}clk_if

board_system_configure-connect{fb1.CLK_IFfb2.CLK_IF}

board_system_save -board haps_80d.vb

```

### Example 2 - I/O Connectors

```

board_system_create -haps -namemysystem

board_system_create -add HAPS80D_DUAL -namefb1 -speed_grade {-1-
c}
board_system_create -add HAPS80D_DUAL -name fb2 -speed_grade {-
1-c}

board_system_configure -top_io {fb1.A8}
board_system_create -interconnect -manual CON_CABLE_100_HT3 -
name
fb1_a1_fb1_b1 -connector {fb1.A5 fb2.A5}

board_system_configure-clock{fb1.GCLK1}pll
board_system_configure -voltage {fb1.A4} 1.2
board_system_configure -function {fb1 gpio2 fb1.pmod
fb2.coresight20} true board_system_save -board haps_vu.vb

```

### Example 3 - GCLKS

```

board_system_create -haps -name mysystem
board_system_create -add HAPS80D_DUAL -name fb1
board_system_create -add HAPS80D_DUAL -name fb2

board_system_configure -top_io {fb1.A4 fb1.B4 fb2.A4}

board_system_create -interconnect -manual CON_CABLE_100_HT3 -
name conn1
-connector {fb1.A11 fb1.B11}
board_system_create -interconnect -manual CON_CABLE_100_HT3 -
name conn2
-connector {fb1.B10 fb2.A11}

```

```
board_system_create -interconnect -manual CON_CABLE_100_HT3 -
name conn3
-connector {fb1.B9 fb2.A10}
board_system_create -interconnect -manual CON_CABLE_100_HT3 -
name conn4 -connector {fb1.B8 fb2.A9}

board_system_configure -clock fb1.GCLK1 fpga
board_system_configure -clock fb1.GCLK2 fpga
board_system_configure -clock fb1.GCLK3 fpga
board_system_configure -clock fb1.GCLK4 fpga
board_system_configure -clock fb1.GCLK5 fpga
board_system_configure -clock fb1.GCLK6 fpga

board_system_configure -clock fb1.CLK_IF5 fpga
board_system_configure -clock fb1.CLK_IF6 fpga

board_system_configure -clock fb2.GCLK1 fpga
board_system_configure -clock fb2.GCLK2 fpga
board_system_configure -clock fb2.GCLK3 fpga
board_system_configure -clock fb2.GCLK4 fpga

board_system_configure -clock fb2.GCLK11 clk_if
board_system_configure -clock fb2.GCLK12 clk_if

board_system_configure -connect {fb1.CLK_IF fb2.CLK_IF}
```

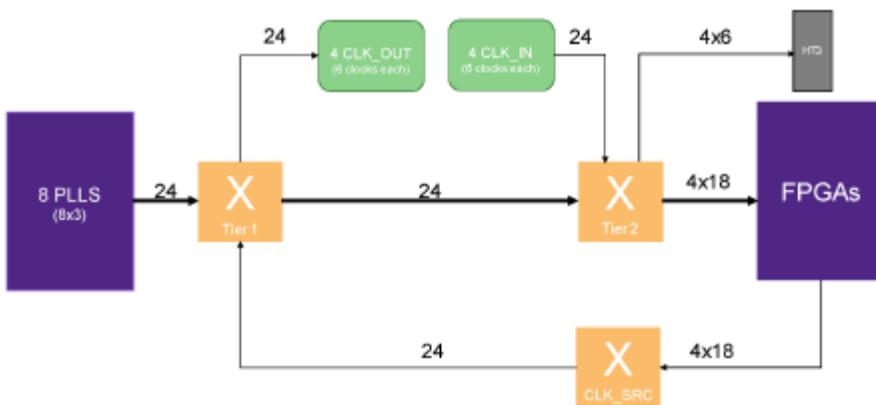
# Working with HAPS-100 Modules

See the following for setup information for HAPS-100 modules:

- [Working with HAPS-100 Clocks](#), on page 296
- [Generating Controlled Clocks using HAPS Clock Generator](#), on page 304
- [Working with HAPS-100 Resets](#), on page 318
- [Using UMRBus 3.0 with HAPS-100 Modules](#), on page 455

## Working with HAPS-100 Clocks

Conceptually, HAPS-100 has the same clock architecture as HAPS-80, but with more local PLL sources and all the clock connectors are equivalent. Each FPGA independently chooses 18 of 48 possible clocks. GCLKs are not shared in multi-design deployments. Any FPGA can source any global clock.



See these topics for details:

- [Specifying HAPS-100 Clock Descriptions in the TSS File](#), on page 297
- [HAPS-100 Clock Example](#), on page 298
- [Using CLK\\_SRC for Clock Connections](#), on page 297
- [Specifying PLL Clocks for HAPS-100 Modules](#), on page 300
- [HAPS-100 Clock Example](#), on page 298

- [PLL and FPGA Source Clock Example for HAPS-100, on page 302](#)
- [Specifying FPGA Clocks for HAPS-100 Modules, on page 303](#)
- [Two-System FPGA Clock Example for HAPS-100, on page 303](#)
- [Five-System FPGA Clock Example for HAPS-100, on page 305](#)

## Specifying HAPS-100 Clock Descriptions in the TSS File

1. The following TSS commands request clock sources and route them to FPGAs:

```
#Create and name clock sources
board_system_configure -clk_src {FB1.PLL} -name cpuClk -frequency
12000
board_system_configure -clk_src {FB1.UA.CLK_SRC11} -name busClk

#Route clock sources to available clock pins
board_system_configure -clock {FB1.u}-name busClk
board_system_configure -clock {FB1.UA FB1.UB}-name cpuClk

#Name specific clock sources
board_system_configure -clock {FB1.UB.CLK3} -name cpuClk
```

2. The TSS automatically routes through clock cables.

```
board_system_configure -clk_src {FB1.PLL} -name cpuClk -frequency
12000
board_system_configure -clock {FB1.UA} -name cpuClk

#Clock cable mounted between FB1.CLKOUT1 and FB2.CLKIN3
board_system_configure -clock {FB2.u} -name cpuClk
```

See the example that follows: [HAPS-100 Clock Example, on page 298](#).

## Using CLK\_SRC for Clock Connections

The following tss and pcf syntax shows how to define FPGA clock connections using CLK\_SRC to reset on a HAPS-100 module.

1. Get the trace name for the clock source using report target\_system.
2. Use the TSS and PCF commands shown below to define the clock connections.

## TSS Syntax

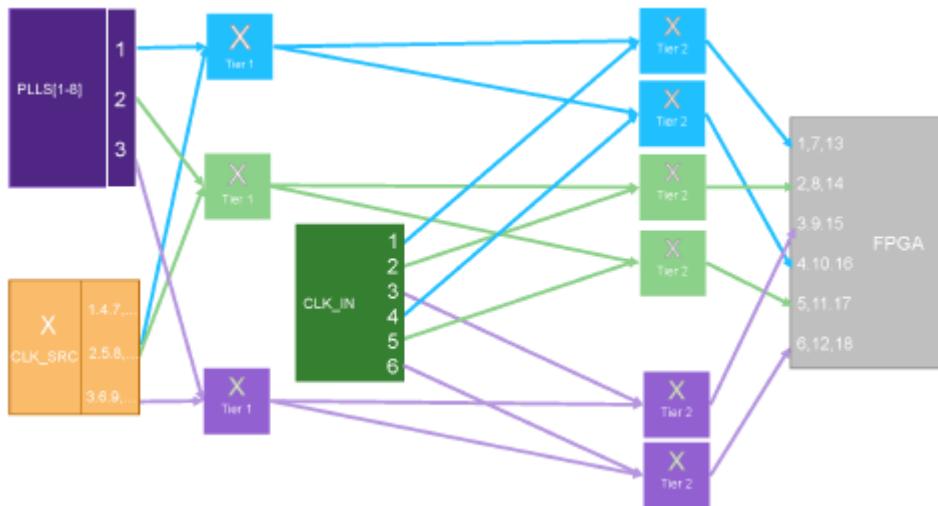
```
board_system_configure -clk_src {fb1.uA.CLK_SRC2}
-name ccm_clk_0 -frequency 5000
board_system_configure -clock {fb1.uA.CLK5} ccm_clk_0
board_system_configure -clock {fb1.uB.CLK5} ccm_clk_0
board_system_configure -clock {fb1.uC.CLK5} ccm_clk_0
board_system_configure -clock {fb1.uD.CLK5} ccm_clk_0
```

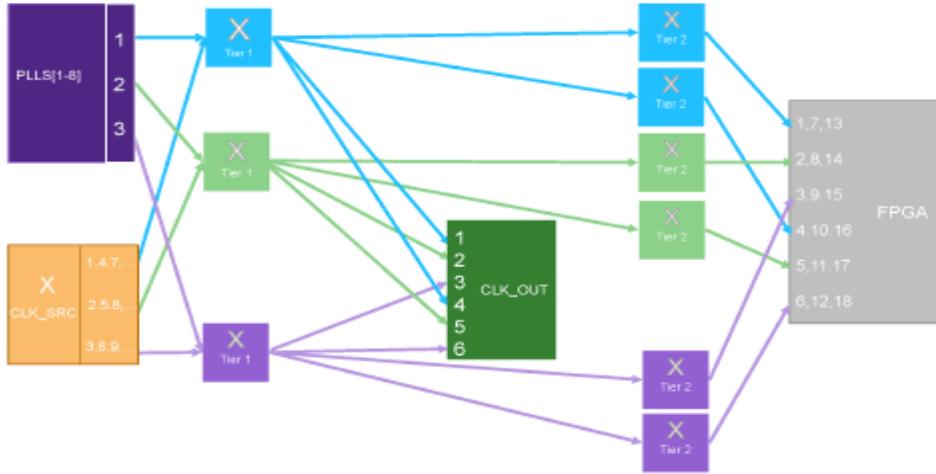
## PCF Syntax

```
assign_port {ccm_out1} -trace {A_CLK_SRC[2]}
assign_global_net clk_ccm_pll ccm_clk_0
```

## HAPS-100 Clock Example

The following illustrations show valid clock combinations. CLK1, 4, 7, 10, 13, and 16 of each FPGA can be connected to the CLK1 output of any of the eight available PLLs.





Valid combinations for the remaining PLL and FPGA generated clocks are shown in the table below.

|             | Sources                    | Valid Sinks                                     |
|-------------|----------------------------|-------------------------------------------------|
| PLL Clocks  | PLL[1-8].CLK1              | FPGA.CLK1,4,7,10,13,16<br>CLK_OUT[1-4].CLK[1,4] |
|             | PLL[1-8].CLK2              | FPGA.CLK2,5,8,11,14,17<br>CLK_OUT[1-4].CLK[2,5] |
|             | PLL[1-8].CLK3              | FPGA.CLK3,6,9,12,15,18<br>CLK_OUT[1-4].CLK[3,6] |
| FPGA Clocks | CLK_SRC1, 4, 7, 10, 13, 16 | FPGA.CLK1,4,7,10,13,16<br>CLK_OUT[1-4].CLK[1,4] |
|             | CLK_SRC2, 5, 8, 11, 14, 17 | FPGA.CLK2,5,8,11,14,17<br>CLK_OUT[1-4].CLK[2,5] |
|             | CLK_SRC3, 6, 9, 12, 15, 18 | FPGA.CLK3,6,9,12,15,18<br>CLK_OUT[1-4].CLK[3,6] |
| CLK_IN      | CLK_IN[1-4].CLK[1,4]       | FPGA.CLK1,4,7,10,13,16                          |
|             | CLK_IN[1-4].CLK[2,5]       | FPGA.CLK2,5,8,11,14,17                          |
|             | CLK_IN[1-4].CLK[3,6]       | FPGA.CLK3, 6, 9, 12, 15, 18                     |

|             | <b>Sinks</b>          | <b>Valid Sources</b>                                                   |
|-------------|-----------------------|------------------------------------------------------------------------|
| FPGA Clocks | FPGA.CLK1,7,13        | PLL[1-8].CLK1<br>CLK_IN[1-4].CLK1<br>CLK_SRC1, 4, 7, 10, 13, 16        |
|             | FPGA.CLK2,8,14        | PLL[1-8].CLK2<br>CLK_IN[1-4].CLK2<br>CLK_SRC2, 5, 8, 11, 14, 17        |
|             | FPGA.CLK3,9,15        | PLL[1-8].CLK3<br>CLK_IN[1-4].CLK3<br>FPGA.CLK_SRC[3, 6, 9, 12, 15, 18] |
|             | FPGA.CLK4,10,16       | PLL[1-8].CLK1<br>CLK_IN[1-4].CLK4<br>CLK_SRC1, 4, 7, 10, 13, 16        |
|             | FPGA.CLK5,11,17       | PLL[1-8].CLK2<br>CLK_IN[1-4].CLK5<br>CLK_SRC2, 5, 8, 11, 14, 17        |
|             | FPGA.CLK6,12,18       | PLL[1-8].CLK3<br>CLK_IN[1-4].CLK6<br>FPGA.CLK_SRC[3, 6, 9, 12, 15, 18] |
| CLK_OUT     | CLK_OUT[1-4].CLK[1,4] | PLL[1-8].CLK1<br>FPGA.CLK_SRC[1,4,7,10,13,16]                          |
|             | CLK_OUT[1-4].CLK[2,5] | PLL[1-8].CLK2<br>FPGA.CLK_SRC[2, 5, 8, 11, 14, 17]                     |
|             | CLK_OUT[1-4].CLK[3,6] | PLL[1-8].CLK3<br>FPGA.CLK_SRC[3, 6, 9, 12, 15, 18]                     |

| <b>Bank</b> | <b>Clocks</b>            |
|-------------|--------------------------|
| 69          | CLK1-3, CLK10, CLK13- 16 |
| 70          | CLK4-6, CLK11            |
| 71          | CLK7-9, CLK12            |

## Specifying PLL Clocks for HAPS-100 Modules

Each HAPS-100 FPGA has independent clocks. The FPGAs that the clocks connect across must also be specified.

1. Set the frequency for HAPS-100.
2. Specify the connections across FPGAs.

In this case, myclk1 can be connected across 1, 2, 3, or 4 FPGAs. The following code snippet shows the connection across all 4 FPGAs.

```
board_system_configure -clk_src {PLL} -frequency 1000 -name myclk1
board_system_configure -clock {FB1.UA.CLK1} myclk1
board_system_configure -clock {FB1.UB.CLK1} myclk1
board_system_configure -clock {FB1.UC.CLK1} myclk1
board_system_configure -clock {FB1.UD.CLK1} myclk1
```

3. Assign the clock to a specific PLL output.

```
board_system_configure -pll fb1.PLL1.CLK1 -frequency 1000 -name myclk1
board_system_configure -clock {FB1.UA.CLK1} myclk1
board_system_configure -clock {FB1.UB.CLK1} myclk1
board_system_configure -clock {FB1.UC.CLK1} myclk1
board_system_configure -clock {FB1.UD.CLK1} myclk1
```

See the following examples: [PLL Source Clock Example, on page 301](#) and [PLL and FPGA Source Clock Example for HAPS-100, on page 302](#).

## PLL Source Clock Example

In the following generic format example a PLL is specified as the source for the clocks and they are connected across all the FPGAs in the system.

## TSS Example

```
board_system_create -haps -name DefaultSystem
board_system_create -add HAPS100_4F -name FB1

#Specify cable connections
board_system_create -interconnect -manual CON_CABLE_50_HT3 -name FB1_1 -connector {FB1.A1 FB1.B1}

#Specify top I/O connections
board_system_configure -top_io {FB1.A4}
```

```
#Specify clock connections. In below example, PLL is specified as
source of clocks and they are connected across all FPGAs.
board_system_configure -clk_src {PLL} -frequency 1000 -name myclk1
board_system_configure -clock {FB1.uA.CLK1} myclk1
board_system_configure -clock {FB1.uB.CLK1} myclk1
board_system_configure -clock {FB1.uC.CLK1} myclk1
board_system_configure -clock {FB1.uD.CLK1} myclk1

board_system_configure -clk_src {PLL} -frequency 1000 -name myclk2
board_system_configure -clock {FB1.uA.CLK2} myclk2
board_system_configure -clock {FB1.uB.CLK2} myclk2
board_system_configure -clock {FB1.uC.CLK2} myclk2
board_system_configure -clock {FB1.uD.CLK2} myclk2

#Specify remaining clock connections in same format.
```

## PCF Commands

```
assign_global_net designclk1 myclk1
assign_global_net designclk2 myclk2
```

## PLL and FPGA Source Clock Example for HAPS-100

The following example shows two PLLs and an FPGA specified as sources for the clocks.

## TSS Commands

```
board_system_create -haps -name DefaultSystem
board_system_create -add HAPS100_4F -name FB1

board_system_create -interconnect -auto -width 100 -devices
{FB1.uA FB1.uB}

#Specify cable connections
board_system_create -interconnect -manual CON_CABLE_50_HT3 -name
FB1_1 -connector {FB1.A1 FB1.B1}

#Specify top I/O connectors.
board_system_configure -top_io {FB1.A4}
board_system_configure -top_io {FB1.B4}
```

```

Below 2 clocks with PLL as source. myclk11 and myclk12 are
connected to only FPGA A.
board_system_configure -clk_src PLL -name myclk11 -frequency 1234
board_system_configure -clock FB1.uA.CLK11 myclk11
board_system_configure -clk_src PLL -name myclk12 -frequency 1234
board_system_configure -clock FB1.uA.CLK12 myclk12

#Below clock with FPGA as source.
board_system_configure -clk_src {FB1.uA} -name myclk1 -frequency
1234
board_system_configure -clock FB1.uB.CLK1 myclk1
board_system_configure -clock FB1.uA.CLK1 myclk1

#Specify the remaining pll/fpga clocks in the design in same format
as shown above.

```

## PCF Commands

```

assign_global_net clk_in_1 myclk11
assign_global_net clk_in_2 myclk12
assign_global_net clk_out_1 myclk1

```

## Specifying FPGA Clocks for HAPS-100 Modules

1. Specify the frequency and the FPGAs connected to the FPGA clock.

```

board_system_configure -clk_src {FB1.uA} -name myclk1 -frequency
1234
board_system_configure -clock FB1.uB.CLK1 myclk1
board_system_configure -clock FB1.uA.CLK1 myclk1

```

2. For clock synchronization, specify where the clock cables are connected.

```
board_system_configure -connect {FB1.CLK_OUT1 FB2.CLK_IN1}
```

See the following examples: [Two-System FPGA Clock Example for HAPS-100, on page 303](#) and [Two-System FPGA Clock Example for HAPS-100, on page 303](#).

## Two-System FPGA Clock Example for HAPS-100

### TSS Commands

```

board_system_create -haps -name DefaultSystem
board_system_create -add HAPS100_4F -name FB1
board_system_create -add HAPS100_4F -name FB2

```

```
#Specify cable connections
board_system_create -interconnect -auto -width 100 -devices
{FB1.uA FB2.uB}

#Specify top I/O
board_system_configure -top_io {FB1.A4}
board_system_configure -top_io {FB2.B4}

#Specify clocks with PLL as source
board_system_configure -clk_src PLL -name myclk11 -frequency 1234
board_system_configure -clock FB1.uA.CLK11 myclk11
board_system_configure -clk_src PLL -name myclk12 -frequency 1234
board_system_configure -clock FB1.uA.CLK12 myclk12

#Specify clocks with FB1.PLL as source
board_system_configure -clk_src FB1.PLL -name pllclk -frequency
1234

#Specify clocks with FB2.uA as source
board_system_configure -clk_src FB2.uA.CLK_SRC2 -name fpgaclk
-frequency 1234

pllclk sourced from any PLL clock from FB1 will be driving FB2
all fpga CLK1 pin
board_system_configure -clock FB2.uA.CLK1 pllclk
board_system_configure -clock FB2.uB.CLK1 pllclk
board_system_configure -clock FB2.uC.CLK1 pllclk
board_system_configure -clock FB2.uD.CLK1 pllclk

fpgaclk sourced from FB2 FPGA A will be driving FB1 all fpga CLK2
pin
board_system_configure -clock FB1.uA.CLK2 fpgaclk
board_system_configure -clock FB1.uB.CLK2 fpgaclk
board_system_configure -clock FB1.uC.CLK2 fpgaclk
board_system_configure -clock FB1.uD.CLK2 fpgaclk

Connect CLK_IN and CLK_OUT connectors on both system
board_system_configure -connect [list FB1.CLK_OUT1 FB2.CLK_IN1]
board_system_configure -connect [list FB1.CLK_OUT2 FB2.CLK_IN2]
board_system_configure -connect [list FB1.CLK_OUT3 FB2.CLK_IN3]
board_system_configure -connect [list FB1.CLK_OUT4 FB2.CLK_IN4]
```

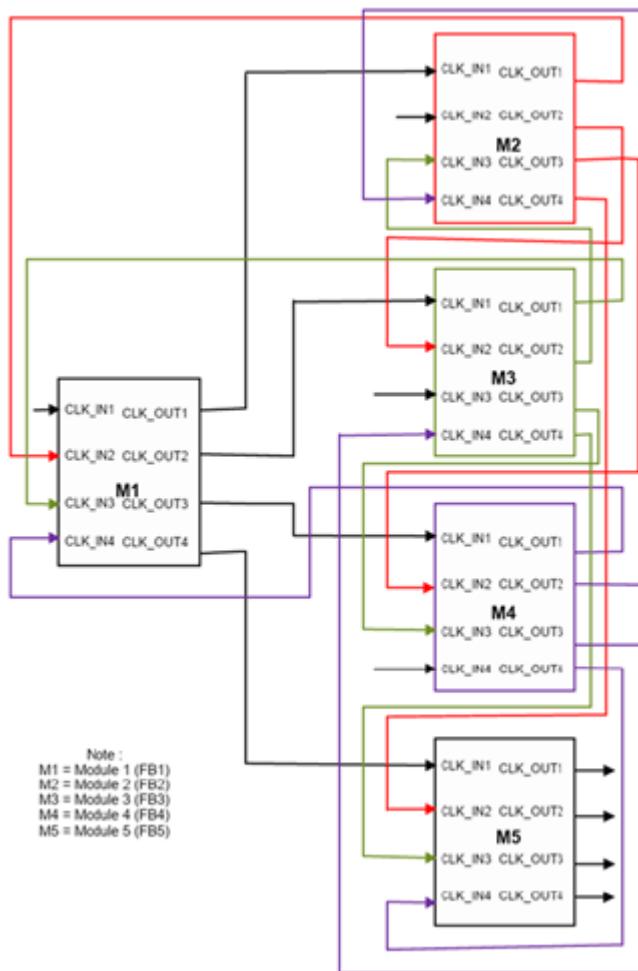
## PCF commands

```
assign_global_net aclk1 myclk11
assign_global_net aclk2 myclk12
assign_global_net userpllclk pllclk
assign_global_net userfpclk fpgaclk
```

The PCF commands are the same format as the prior examples.

## Five-System FPGA Clock Example for HAPS-100

This example shows clock synchronization using CLK\_IN and CLK\_OUT. Six clocks are generated from FB1-4 resulting in a total of 24 clocks, of which 18 clocks can go to each FPGA. Then the FPGAs to which the clocks are connected are specified. In this example, 12 clocks are connected across 20 FPGAs.



## TSS Commands

```
board_system_create -haps -name DefaultSystem
board_system_create -add HAPS100_4F -name FB1
board_system_create -add HAPS100_4F -name FB2
board_system_create -add HAPS100_4F -name FB3
board_system_create -add HAPS100_4F -name FB4
board_system_create -add HAPS100_4F -name FB5
```

```
FB1 generated clocks
board_system_configure -clk_src FB1.PLL -name myclk1 -frequency
10000
board_system_configure -clk_src FB1.PLL -name myclk2 -frequency
10000
board_system_configure -clk_src FB1.PLL -name myclk3 -frequency
10000
board_system_configure -clk_src FB1.PLL -name myclk4 -frequency
10000
board_system_configure -clk_src FB1.PLL -name myclk5 -frequency
10000
board_system_configure -clk_src FB1.PLL -name myclk6 -frequency
10000

#Likewise specify 6 clocks generated from FB2, FB3, and FB4. A
total of 24 clocks of which 18 clocks can go to each FPGA.

#FB2 generated clocks

#FB3 generated clocks

#FB4 generated clocks

#Specify across which FPGAs the clocks are connected. In this
example, 12 clocks are connected across 20 FPGAs.

for {set a 1} {$a < 13} {incr a} {
 board_system_configure -clock FB1.uA.CLK$a myclk$a
 board_system_configure -clock FB1.uB.CLK$a myclk$a
 board_system_configure -clock FB1.uC.CLK$a myclk$a
 board_system_configure -clock FB1.uD.CLK$a myclk$a
 board_system_configure -clock FB2.uA.CLK$a myclk$a
 board_system_configure -clock FB2.uB.CLK$a myclk$a
 board_system_configure -clock FB2.uC.CLK$a myclk$a
 board_system_configure -clock FB2.uD.CLK$a myclk$a
 board_system_configure -clock FB3.uA.CLK$a myclk$a
 board_system_configure -clock FB3.uB.CLK$a myclk$a
 board_system_configure -clock FB3.uC.CLK$a myclk$a
 board_system_configure -clock FB3.uD.CLK$a myclk$a
 board_system_configure -clock FB4.uA.CLK$a myclk$a
 board_system_configure -clock FB4.uB.CLK$a myclk$a
 board_system_configure -clock FB4.uC.CLK$a myclk$a
 board_system_configure -clock FB4.uD.CLK$a myclk$a
 board_system_configure -clock FB5.uA.CLK$a myclk$a
 board_system_configure -clock FB5.uB.CLK$a myclk$a
 board_system_configure -clock FB5.uC.CLK$a myclk$a
 board_system_configure -clock FB5.uD.CLK$a myclk$a
}
```

---

#Specify CLK\_OUT & CLK\_IN connections. FB1 generates 6 clocks, Cables from FB1 to the remaining 4 FPGAs. There are 6 clocks across 5 FPGAs. Likewise, connect cables from FB2, FB3, and FB4 to the remaining FPGAs. A total of 24 clocks across 5 systems of which each FPGA can receive 18 clocks.

```
board_system_configure -connect [list FB1.CLK_OUT1 FB2.CLK_IN1]
board_system_configure -connect [list FB1.CLK_OUT2 FB3.CLK_IN1]
board_system_configure -connect [list FB1.CLK_OUT3 FB4.CLK_IN1]
board_system_configure -connect [list FB1.CLK_OUT4 FB5.CLK_IN1]
board_system_configure -connect [list FB2.CLK_OUT1 FB1.CLK_IN2]
board_system_configure -connect [list FB2.CLK_OUT2 FB3.CLK_IN2]
board_system_configure -connect [list FB2.CLK_OUT3 FB4.CLK_IN2]
board_system_configure -connect [list FB2.CLK_OUT4 FB5.CLK_IN2]
board_system_configure -connect [list FB3.CLK_OUT1 FB1.CLK_IN3]
board_system_configure -connect [list FB3.CLK_OUT2 FB2.CLK_IN3]
board_system_configure -connect [list FB3.CLK_OUT3 FB4.CLK_IN3]
board_system_configure -connect [list FB3.CLK_OUT4 FB5.CLK_IN3]
board_system_configure -connect [list FB4.CLK_OUT1 FB1.CLK_IN4]
board_system_configure -connect [list FB4.CLK_OUT2 FB2.CLK_IN4]
board_system_configure -connect [list FB4.CLK_OUT3 FB3.CLK_IN4]
board_system_configure -connect [list FB4.CLK_OUT4 FB5.CLK_IN4]
```

The PCF commands follow the same format as the previous examples.

## Generating Controlled Clocks using HAPS Clock Generator

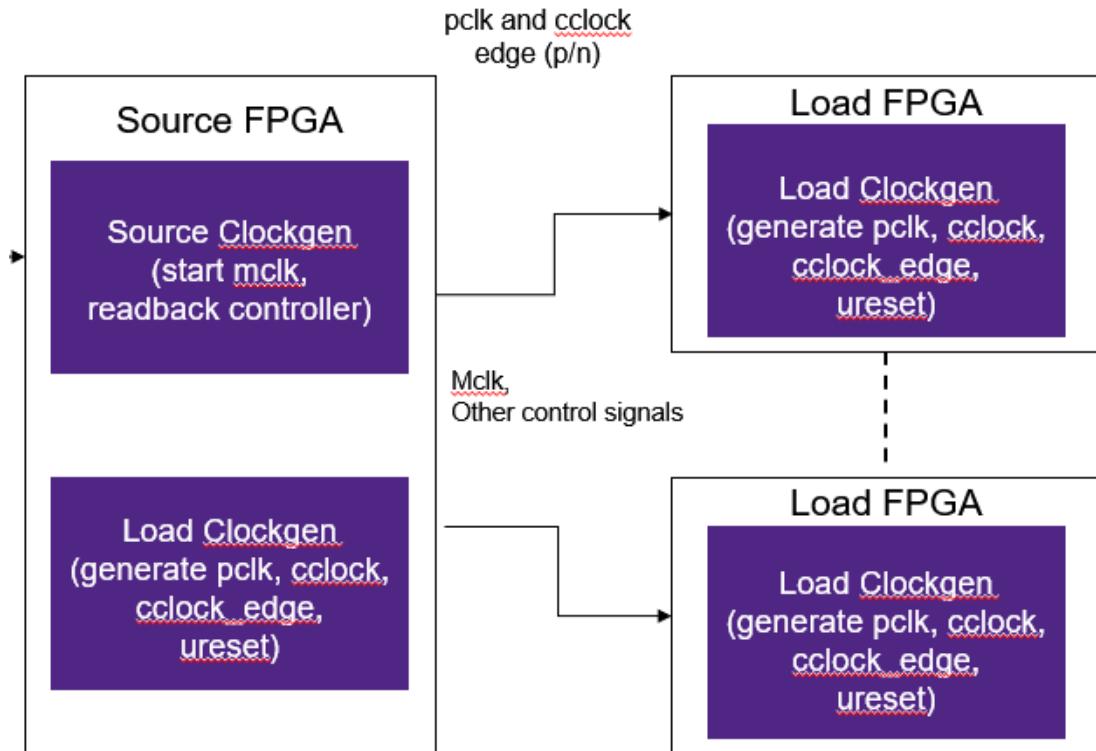
*(HAPS-100 Only)*

The HAPS Clock Generator (HAPS Clockgen) provides emulation clocking support to prototyping designs through the ProtoCompiler tool. Using HAPS Clockgen, all clocks are derived from a single origin clock called the driver clock. These clocks can be turned into controlled clocks whose shape can be programmed during the design run using the Protocompiler RunTime. Additionally, these controlled clocks can be stopped and started during the design run.

Deriving all clocks from a driver clock ensures that gated clock conversion (GCC) never fails if no free running clocks are mixed with a controlled clock.

The driver clock in turn is generated from a fast source clock that is derived from an on-board PLL whose frequency can be set in the TSS. The default source frequency is 100 MHz. The frequency of the source clock must be same or higher than the frequency of the driver clock. This frequency is computed as the minimum of (rise, fall, (period -fall) times) over all controlled

clocks. You must specify one of the FPGAs used in the system setup as the Source FPGA in the TSS file. The source clock from the on board PLL is connected only to the source FPGA and is used in the Source ClockGen module. The source clock is propagated via the GCLK (CLK\_SRC) skew balanced trace into all other FPGAs called load FPGAs and back into the source FPGA. The driver clock is derived in the Load Clockgen as shown in the following image.



The tool ensures that the skew on driver clock and each controlled clock is balanced to less than 4 nsec.

By default, the source clock is not run after bit file configuration. The runtime script needs to explicitly start the source clock using the commands illustrated in [Reconfiguring Clocks in ProtoCompiler Runtime, on page 316](#).

The first positive edge of the source clock is used to initialize the Load Clockgen in each FPGA. The individual controlled clock shape (rise time, fall time and period) can be programmed only before the source clock is started for the first time after bit file configuration. However, note that configuring a

controlled clock shape do not lead to either that controlled clock or driver clock speeding up from the frequencies for which Static Timing Analysis was done, in which case there may be functional failures.

The driver clock period can be configured at any time during the design execution on HAPS-100 platform by stopping the source clock, writing the new driver clock period, and then restarting the source clock. The controlled clock shape parameters (rise time , fall time , period) and the driver clock period at any point during runtime can be queried using the command described in [Reconfiguring Clocks in ProtoCompiler Runtime, on page 316](#).

Additionally, the tool generates a reset called ureset that is synchronous to the driver clock. The tool automatically replicates the clock generator across all FPGAs where controlled clocks are needed and ensures that the corresponding controlled clock and reset on each FPGA remains in phase. The controlled clocks need not have any integer relationship amongst themselves or any restriction on phase offset or period.

Using HAPS Clockgen, you can generate a maximum of 252 clocks. Each clock can be used to generate any number of derived clocks by user logic.

Controlled clocks are inferred using FDC and TSS.

The following hyper\_connects are available to connect various useful signals generated from the load clock generator in each FPGA.

- haps\_uclock—Driver clock
- haps\_mclk—Source clock
- haps\_ureset—Ureset. An active high reset that remains active initially but gets deactivated after the driver clock starts.
- haps\_cclock\_locked—Signal that goes high after all initialization routines in the HAPS Clockgenerator is complete and the cclock waveforms are stable.

## Clockgen Nomenclature

- SourceClock (mclk): Fast constant frequency clock. The only source clock coming from PLL.
- Driver Clock (pclk): Base clock for all sequential elements after GCC.
- Controlled Clock (cclock): Output of the clock generator; replaced by pclk and cclock\_edge after GCC.

- Cclock Edge (cclock\_edge): Active edge predictor of cclock after GCC.

## Adding HAPS Clockgen to the Design

1. Enable HAPS Clockgen using the **option set** commands:

```
option set haps_clockgen_mode 1
```

2. Configure the GCLK network to route the clock, automate the clock connection and set the base clock frequency using TSS:

```
board_system_configure -create_ccm_clocks -source SourceFPGAName -load {all|loadFPGAList} -source_frequency sourceClockFrequencyInKHz
```

Example:

```
board_system_configure -create_ccm_clocks -source mb1.uA -load {mb1.uA mb1.uB}
```

3. Configure clocks using HAPS Clockgen.

The following GCLK traces are reserved for HAPS Clockgen:

- <fpga>. CLK1—reserved for Master clock source
- <fpga>. CLK2—reserved for Readback
- <fpga>. CLK3—reserved for Clockgen reset
- <fpga>. CLK4—reserved for Master Enable

Do not configure these traces for other applications. If you try to connect any of the reserved traces, the tool displays error **CU725** at the pre-partition stage.

Additionally, if you have set `enable_zebu_dpi` 1, ensure that there is at least one available trace from the source FPGA to each load FPGA and one trace from each load FPGA to source FPGA.

4. For mclk, the source tool checks for used PLLs and chooses a free PLL accordingly.
5. Configure constraints using FDC:

```
create_clockgen_group {listOfClocks} -group {groupName}
```

In the following example, clk1 and clk2 are generated by HAPS Clockgen and disconnected from the original clock source:

```
create_clockgen_group {clk1 clk2} -group {grp1}
```

6. Run pre-partition.
7. Verify the Pre-partition report to confirm the clocks are generated.

## Example

The following example uses RTL, FDC, and TSS file inputs to generate clocks. In this example, clk1 and clk2 are generated from HAPS clockgen:

RTL Inputs:

```
module top (
 input clk, clk1, clk2,
 input in, in1, in2,
 output dout, dout1, dout2
);
// reg/ wire
reg dout = 0;
reg dout1 = 0;
reg dout2 = 0;

always@(posedge clk) dout <= in;
always@(posedge clk1) dout1 <= in1;
always@(posedge clk2) dout2 <= in2;
endmodule
```

FDC Inputs:

```
create_clock -name clk [get_ports {p:clk}] -period 30 // clk is a
free running clock that is not handled by the HAPS clockgen. It
cannot mix with the clock cone that contains clk1, clk2 or any
clock / enable derived from that.
create_clock -name clk1 [get_ports {p:clk1}] -period 70 -waveform
{0 35}
create_clock -name clk2 [get_ports {p:clk2}] -period 100 -waveform
{0 50}
create_clockgen_group {clk1 clk2} -group {grp1}
```

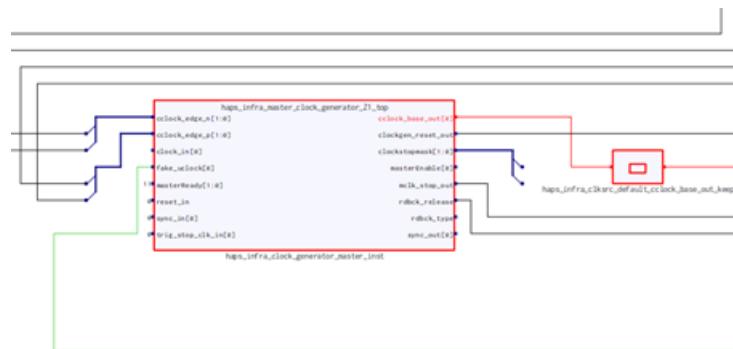
TSS Inputs:

```
board_system_create -haps -name Test_board
board_system_create -add HAPS100_4F -name mb1
board_system_configure -create_ccm_clocks -source mb1.uA -load
{mb1.uA mb1.uB}
```

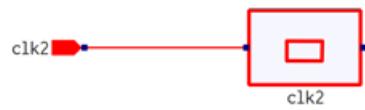
Compile View:



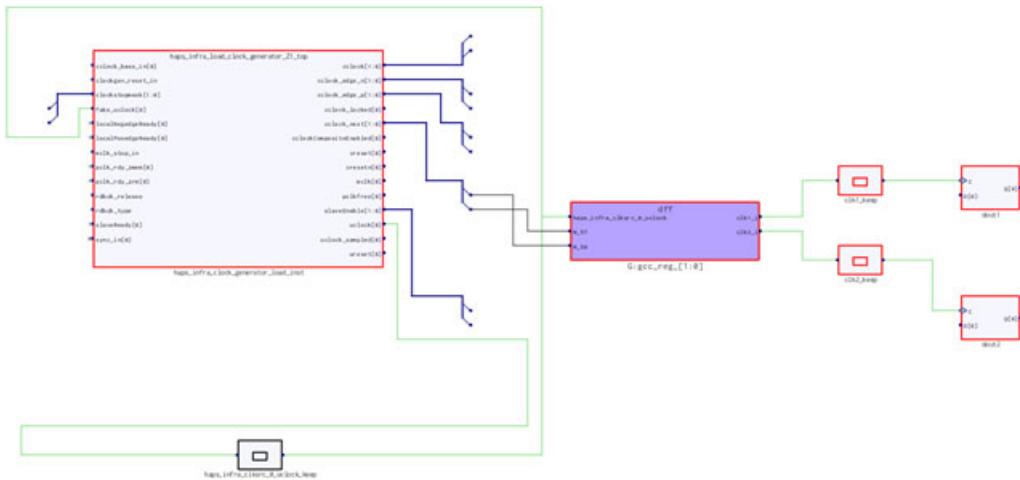
Pre-Partition View—Master Module:



The user controlled clocks are disconnected from the ports and are generated by HAPS Clockgen:



## Clock Generator Load Module:



All controlled clocks are generated from the clockgen.

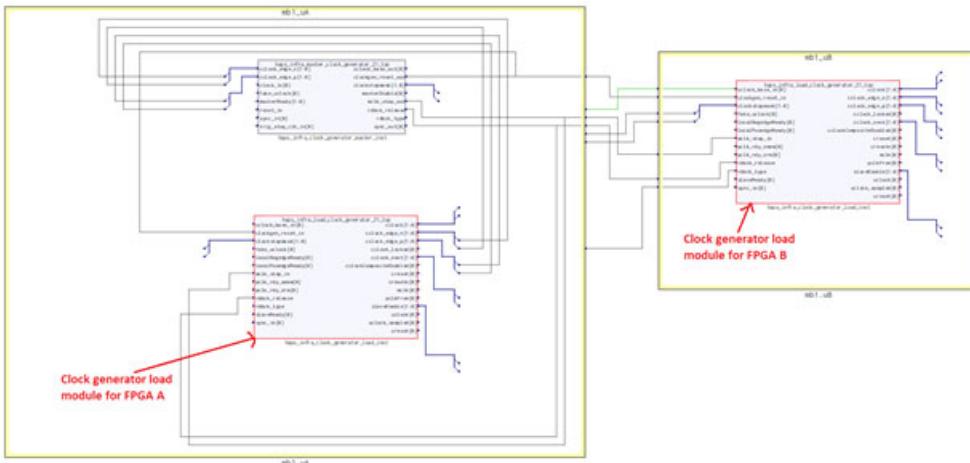
## Pre-Partition Report:

In the pre-partition report, you can see the driver clock and the controlled clocks generated by the driver clock.

| #88 [Clock Summary] Driver_clock (pclk) |                                                                             | Requested Frequency | Requested Period | Clock Type                  |
|-----------------------------------------|-----------------------------------------------------------------------------|---------------------|------------------|-----------------------------|
| Level                                   | Start Clock                                                                 |                     |                  |                             |
| 0 -                                     | driverclk0 (nihaps_infra_cikarc_0_uclock)                                   | 33.3 MHz            | 30.000           | declared                    |
| 1 -                                     | clk1 (tiproc_reg_0_Q[0])                                                    | 16.7 MHz            | 60.000           | generated (from driverclk0) |
| 1 -                                     | clk2 (tiproc_reg_1_Q[0])                                                    | 16.7 MHz            | 60.000           | generated (from driverclk0) |
| 0 -                                     | clk ({gen_ports pclk})                                                      | 33.3 MHz            | 30.000           | declared                    |
| 0 -                                     | hape_infra_cikarc_default_out (nihaps_infra_clksrc_default_oclock_base_out) | 100.0 MHz           | 10.000           | declared                    |
| 0 -                                     | hape_infra_cikarc_default (nihaps_infra_cikarc_default)                     | 100.0 MHz           | 10.000           | declared                    |

## Partition Schematic:

During partition, the load module gets replicated to all FPGAs. In the following figure you can see that the Clockgen Load module gets replicated in FPGA A and FPGA B.



## Reconfiguring Clocks in ProtoCompiler Runtime

(HAPS-100 only)

Using Clock Generator, you can reconfigure clocks at runtime using the **CCM** command. For details, see *HAPS® Prototyping Debugging Environment Reference* manual.

Use the following TCL script to reconfigure controlled clock or driver clock frequency at runtime:

```
package require CCM # calling the clock generator runtime package
CCM package

CCM::open $configid # open the handle to call the clock generator
routines

CCM::enable_source_clock 0 # stops source clock to freeze the
system
CCM::create_clock -name hw_top.clk -period <period in ns> -
waveform [list <rise_time> <fall_time>] # configures controlled
clock period
CCM::dclock_period <period in ns> # configure driver clock period
CCM::apply_config # write overriding values into the system
CCM::query_config # get driver clock and cclock period / rise time
/ fall time values in nsec
CCM::enable_source_clock 1 # enable source clock
```

Use the following TCL script to select readback mode on driver clock, composite clock or cclock, and define edges:

```
Commands to configure the board
set handle [cfg_open $systemName1]
cfg_project_clear $handle
cfg_project_configure $handle runtime/system/targetsystem_ab.tsd
puts "Configuration Done"
cfg_close $handle

Commands to reconfigure the clocks in runtime
puts "configuring CCM"
 CCM::enable_source_clock 0
 CCM::create_clock -name top.clk1 -period 200 -waveform [list 0
100]
 CCM::dclock_period 50 ## To configure Driver clock on runtime
 CCM::apply_config
 CCM::query_config
}
Commands to dump the clock counter values
proc dumpCnt {} {
 CCM::enable_source_clock 0
 foreach c [CCM::cclock_names] {
 foreach f [CCM::fpga_names] {
 puts "master clock : ${f}:[CCM::source_clock_counter $f]"
 puts "driver clock : ${f}:[CCM::driver_clock_counter $f]"
 puts "cclock : ${f}_${c}:[CCM::cclock_counter $c $f]"
 }
 }
 CCM::enable_source_clock 1
}
#####
Running the DRC check

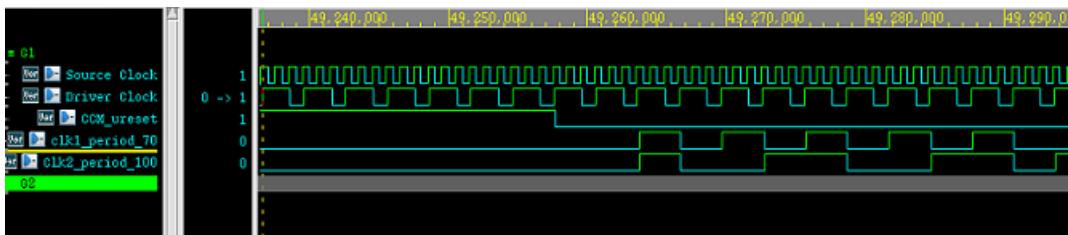
```

```
dumpCnt
after 1000
CCM:::run_drc
```

Use the following TCL script to select readback stepping mode on driver clock, composite clock or cclock, and define edges:

```
#####Performing Synchronous Multi-clock Readback #####
lappend modes "-composite" # Composite clock would display
positive edge when any of the cclocks display a positive edge and
negative edge, when any of the cclocks display a negative edge.
lappend modes "-pos_cclock_id [CCM:::cclock_id hw_top.clk]"
lappend modes "-neg_cclock_id [CCM:::cclock_id hw_top.clk]"
lappend modes "-driver"
foreach mode $modes {
 eval "ccm stepping_mode $mode"
 foreach step [list 1 2 3 4 5 6 7 8 9] {
 run -wait -wait_for_trigger capim -release_clock yes -readback 10
 -stepping $step
 write fsdb my[string map {- _ " " _} ${mode}]_${step}.fsdb
 }
}
```

## } Clock Generation Waveform



## Working with HAPS-100 Resets

See the following topics for information about working with HAPS-100 resets

## Assigning Resets for HAPS-100 Modules

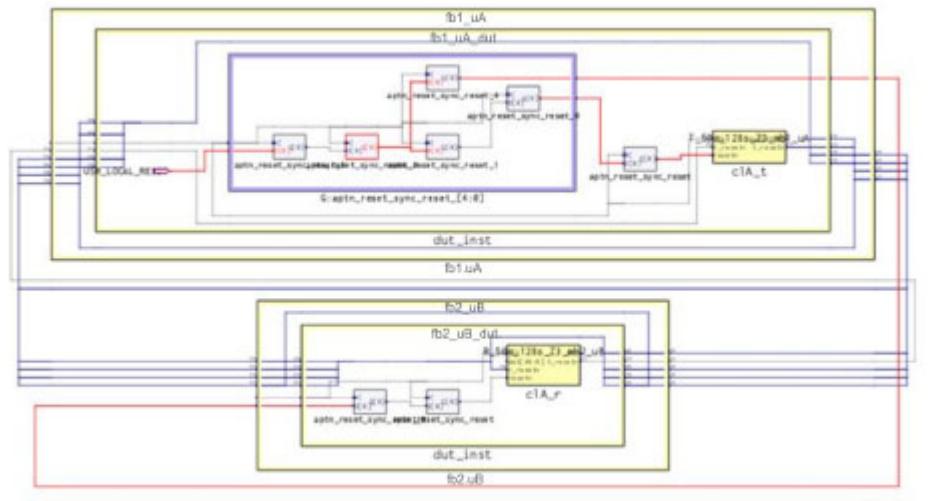
Design reset connections from the system IP. The following pcf syntax is for setting up the design reset connections from the System IP.

```
assign_virtual_port -port reset -type USR_LOCAL_RESET -bin {fb1.uA}
reset_synchronize -net reset -init 0
-extras_pipeline_stages 0 -clock clk
```

## Working with Resets in a Mixed Setup with HAPS-100 and HAPS-80

Use this procedure to synchronize resets across all systems in a mixed design setup with HAPS-100 and HAPS-80 designs.

1. For a reset from a HAPS-100 module, use the reset from FPGA A on HAPS-100 to synchronize across all systems.



Use PCF commands as shown:

```
assign_virtual_port -port reset -type USR_LOCAL_RESET -bin {fb1.uA}
reset_synchronize -net reset -init 0 -extra_pipeline_stages 0 -clock clk
```

2. For a reset from a HAPS-80 system, use the reset from FPGA B on HAPS-80 to synchronize across all systems.

Use PCF commands as shown:

```
net_attribute {reset} -function SINGLE_FPGA_RESET
assign_port {reset} -trace {fb2.B_USER_RESETN}
reset_synchronize -net reset -init 0 -extra_pipeline_stages 0 -clock clk
```

## Using UMRBus 3.0 with HAPS-100 Modules

See these topics for information about using UMRBus 3.0 with HAPS-100:

- [Using UMRBus 3.0 MCAPIM for HAPS-100 Modules](#), on page 455
- [Using UMRBus 3.0 Transactors](#), on page 457
- [Using JTAG Communication](#), on page 458

### Using UMRBus 3.0 MCAPIM for HAPS-100 Modules

Only UMRBus 3.0 MCAPIM (`umr3_mcapim_ui`) is supported for HAPS-100 modules. It allows bi-directional communication, with a higher bit width. These are the guidelines for using it:

- For mixed systems, partition UMRBus 3.0 MCAPIM to HAPS-100 and UMRBus 2.0 to HAPS-80.
- UMRBus 3.0 is backward compatible; replacing prior CAPIMs with the new ones. However new features, such as DMA, are not backward compatible.
- Do the following to convert HAPS-80 designs that use UMRBus 2.0 CAPIMs (`capim_ui` or `capim_wp`) to UMRBus 3.0 MCAPIM (`umr3_capim_ui`) for HAPS-100.
  - For a simple `capim_ui`, replace it with `umr3_capim_ui` and connect the UMRBus 3.0 ports. See the example that follows.
  - Remove or comment the parameter `//.UMR_USE_LOCATION_ID(1)`.
- You can downgrade the UMRBus 2.0 error to a warning by setting the environment variable `disable_umr2_capim_error` to TRUE.

#### Example: UMRBus 2.0 to UMRBus 3.0

The following is an example of converting UMRBus 2.0 to UMRBus 3.0. The necessary changes are in bold.

UMRBus 2.0

```
capim_ui #(.UMR_CAPIM_ADDRESS(UMR_CAPIM_BASE_ADDRESS), (
 .UMR_CAPIM_TYPE(16'hD000),
 .UMR_DATA_BITWIDTH(UMR_DATA_BITWIDTH),
 .UMR_USE_LOCATION_ID(1), .UMR_CAPIM_COMMENT_STRING("I_CAPIM_1"))

capim_led_control (
 .umr_clk (umr_clkout),
 .umr_reset (),
 .wr(capim_data_wr),
 .dout(capim_data_do),
 .rd (capim_data_rd),
 .din(capim_data_di),
 .intr(1'b0),
 .inta (),
 .inttype(16'h0000)
);

 .din(capim_data_di),
 .intr(1'b0),
 .inta (),
 .inttype(16'h0000)
);

capim_ui #(.UMR_CAPIM_ADDRESS(UMR_CAPIM_BASE_ADDRESS),
 .UMR_CAPIM_TYPE(16'hD000),
 .UMR_DATA_BITWIDTH(UMR_DATA_BITWIDTH),
 .UMR_USE_LOCATION_ID(1),
 .UMR_CAPIM_COMMENT_STRING("I_CAPIM_1")
)

capim_ua

 .umr_clk(umr_clk),
 .umr_reset(umr_reset),
 .wr(capim_data_wr),
 .dout(capim_data_do),
 .rd(capim_data_rd),
 .din(capim_data_di_ba),
 .intr(1'b0),
 .inta(),
 .inttype(16'h0000));
```

UMRBus 3.0

```

umr3_capim_ui #(.UMR_CAPIM_ADDRESS(UMR_CAPIM_BASE_ADDRESS),
 .UMR_CAPIM_TYPE(16'hD000),
 .UMR_DATA_BITWIDTH(UMR_DATA_BITWIDTH),
 //.UMR_USE_LOCATION_ID(1),
 .UMR_CAPIM_COMMENT_STRING("I_CAPIM_1")
)

capim_ua (
 .umr_clk(umr_clk),
 .umr_reset(umr_reset),
 .wr(capi_dout_valid),
 .dout(capi_dout),
 .rd(),
 .din(capi_din),
 .intr(1'b0),
 .inta(),
 .inttype(16'h0000));

```

## Using UMRBus 3.0 Transactors

Follow these steps to convert existing HAPS-80 transactor designs to HAPS-100. If you use a new transactor design that is implemented for HAPS-100, XACTORS\_GEN automatically sets the right parameters and these steps are not required.

1. Add the define XACTORS\_UMR3 Verilog define as compile option.
2. Add following file to file list

\$env(BUILD)/lib/synip/uc2/umrbus3/umrbus3.v

3. Comment out the umr\_clk\_reset instance in the top-level file.

```

// umr_clk_reset
// I_umr_clk_reset
//(
// .umr_clk_out(CCLOCK),
// .umr_reset_out(CRESET)
//);

```

## Using JTAG Communication

There is no dedicated JTAG connector on the HAPS-100 module. Instead the built-in UMRBus offers JTAG communication for your application. The following two step process uses the virtual cable (XVC) from Vivado Hardware Manager, which is a TCP/IP-based protocol that acts like a JTAG cable.

1. Start the JTAG server using Confpro commands on the host computer.

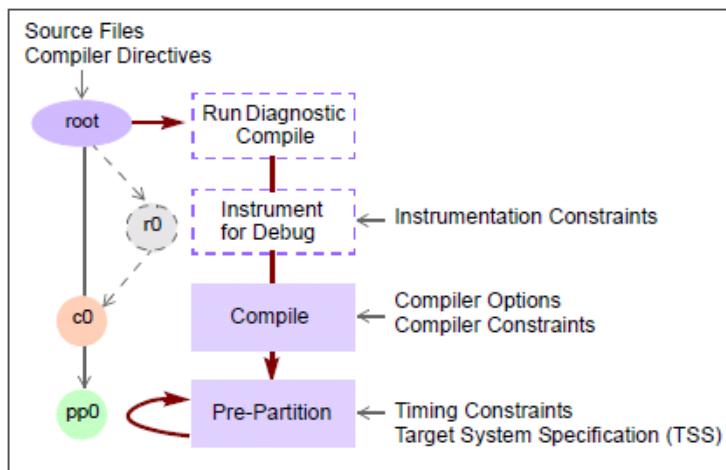
```
> cfg_jtag_start cfg0 fb1.ua [-port 65137]
> cfg_jtag_stop cfg0 fb1.ua
```

2. Connect to JTAG using the Vivado Hardware Manager.

```
>connect_hw_server
>open_hw_target -xvc_url localhost: 65137
```

## Using run pre\_partition to Define Partitions

After you have set up your design and the files are ready ([Setting up Files for Partitioning, on page 183](#)), you can run the preparatory implementation steps before you partition the design. During pre-partitioning, the tool reads target system specifications (TSS), estimates area and applies constraints, including UPF constraints. If the design was instrumented, it also reads the idc file and inserts IICE™ logic. The following figure summarizes these steps and the corresponding database states generated. Dotted lines indicate optional steps.



1. Compile the design.
  - Optionally, run the diagnostic compiler (`report rtl_diagnostics`) and make sure your RTL is clean.
  - If you want to instrument the design, use the `run pre_instrument` command to add RTL debug instrumentation.
  - Compile the design (`run compile`).
2. Create a board specification file, as described in [Defining the Target System in a Detailed TSS File, on page 189](#).

If you are exploring partitioning options, use a basic TSS file for quicker turnaround times. See [Exploring Partition Choices with a Basic TSS, on page 185](#) for information about this file.

3. Set constraints and options as needed in fdc and Tcl options files respectively. Source the Tcl file.

This is the first stage where FDC synthesis constraints and attributes are honored, so make sure to include this file. You can add additional FDC constraints at later stages.

4. Run prepartition:

```
run pre_partition -tss boardFile.tcl
```

- You must specify the TSS board file.
- The tool runs pre-partition with built-in area estimation (fast area estimation) when set to **0**. To run regular area estimation run pre-partitioning with **-area\_est 1**. Area estimation is critical to creating efficient partitions.

To use previously generated area estimates for an incremental run, first run pre-partition with **-out** to generate the database. For the incremental run, set **-area\_est 0**, and specify the previous database.

See [Working with Area Estimates for Partitioning, on page 337](#) for details about using area estimation.

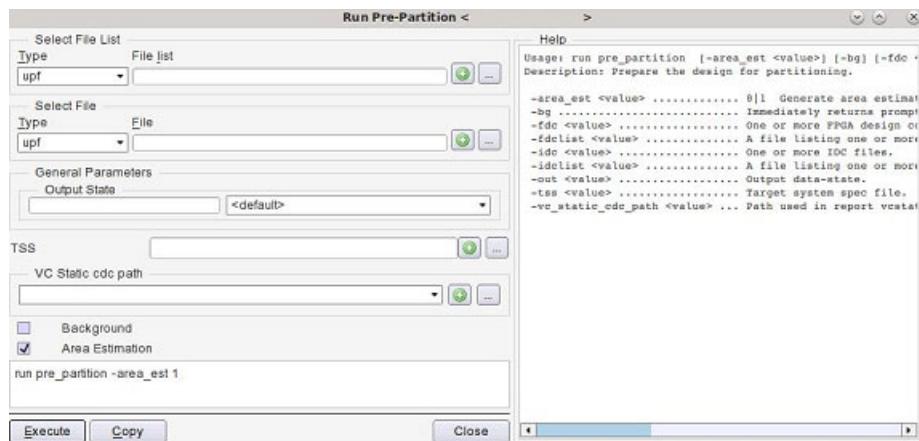
- For more efficient partitioning, set an effort level for area estimation with the option set **area\_effort** command. The default setting is low, but you can set it to medium or high.
- The default number of jobs you can run in parallel is eight, but you can increase that number with the option set **max\_parallel\_jobs** command.
- Specify a pcf file that contains partitioning constraints. For example, set **cell\_utilization** constraints in the pcf file to update area numbers for each instance.
- If you have fdc constraint files, add them at the command line with the **-fdc** argument.

For other options to the run pre\_partition command, refer to the *Command Reference Manual*.

The first time through, the tool runs area estimation. It honors the constraints and options while performing technology-independent optimizations. If you instrumented the design, it inserts debug IICE logic. It creates a new database state (**pp**) that is ready for partitioning by eliminating feedthroughs, duplicate output, unconnected or no-load top-level input ports, and disabled tristates (connected to constants).

The command generates an area estimate file called *design.est* and a compiled netlist (srs) that you can view with the view schematic command. To view the estimate file, use report list and select the area estimate file. You can also export this file for use during the partitioning phase. After running pre-partitioning, the tool generates a new, pre-partitioning database state.

You can also pre-partition using the GUI instead of the command line. Start with a compiled database, select State->Run Pre-Partition, and specify the command parameters. For descriptions of the dialog box options, see [Run Partition Dialog Box, on page 109](#) in the *Reference Manual*.



## 5. Analyze the results.

- Check FDC constraints by running the `report constraint_check` command to generate the `constraint_check.rpt` file. Check this file for details about the constraints that were applied, not applied, or ignored. This command also generates a `syntax_constraint_check.rpt` file which reports any syntax errors with the constraints. To run just the constraint syntax check, use the `report constraint_check -syntax_only` command.

- Use the report list and export report commands to view the reports generated after partitioning. These are the most important reports to check at this stage: Area Estimation Report, Clock Summary, Constraint Application Report, UPF Report, and Netlist Editing Report.

For more information about analysis, see [Analyzing Partition Result Files, on page 358](#) and [Analyzing Results, on page 512](#).

## 6. Make adjustments to options and constraints and rerun the command.

Pre-partitioning is iterative and you can run it as many times as you need. After the initial run, the tool does not rerun area estimation, so subsequent runs take less time than the first run.

- Check the channels and make feasible channels that yield a good TDM ratio, where the total number of pins per device does not exceed the maximum.
- To lock results from a previous run, copy the appropriate assignments from the report into the pcf file for the next run.

# Partitioning the Logic

Partitioning logic is an iterative process, where you refine results until you get what you want. Your goal is to find a feasible partition that meets your minimum timing goals, so that the connections between bins can be routed using the trace groups specified in the TSS file.

The tool automates this difficult task by helping you find a good hardware topology and logic assignment for your design. The hardware configuration determines the functioning of the prototype, but design characteristics determine the best hardware configuration. The partitioning solution described here provides a flexible solution that lets you quickly iterate through high-capacity designs and partition them.

Partitions are implemented with the run partition, run system\_route, and run system\_generate commands. The following topics focus on the run partition command:

- [Using run partition to Create Partitions](#), on page 329
- [Partitioning Single-FPGA Designs](#), on page 335
- [Working with Area Estimates for Partitioning](#), on page 337
- [Using Timing-Aware Partitioning](#), on page 341
- [Optimizing Multi-Hop Paths](#), on page 344
- [Partitioning Clocks](#), on page 348
- [Locking Down Partitions](#), on page 352
- [Using Interactive Partitioning](#), on page 355

For additional partitioning-related information, see these topics:

- [Using run pre\\_partition to Define Partitions](#), on page 324
- [Analyzing Partition Result Files](#), on page 358
- [Using Time Domain Multiplexing \(TDM\)](#), on page 372
- [Using Hierarchical Partitioning](#), on page 406
- [Running System Route for the Top Level](#), on page 412
- [Generating FPGAs](#), on page 416

## Using run partition to Create Partitions

After pre-partitioning, partitioning solutions are explored and implemented with the run partition, run system\_route, and run system\_generate commands. This procedure summarizes how to generate hardware-aware partitions for a multi-FPGA design with the run partition command. The partitions created may be exploratory or final, depending on the TSS hardware definitions and the assignments in the PCF file.

To use hardware-based partitioning for a single-FPGA design, refer to the procedure in [Partitioning Single-FPGA Designs, on page 335](#).

1. Start from a pre-partitioned database state.

See [Using run pre\\_partition to Define Partitions, on page 324](#) for details about generating this state.

If you are working iteratively and have already partitioned and implemented your FPGA partitions, you can refine top-level partitioning by using area estimates generated after implementing the FPGAs. See [Updating the Top Level with FPGA Implementation Results, on page 435](#) for details.

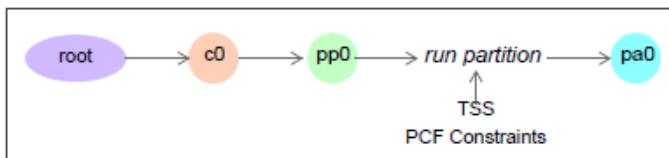
2. Prepare the input files for partitioning.

The files are specified as part of the command line in the next step.

- Describe the hardware setup in a TSS board specification file. (See [Defining the Target System in a Detailed TSS File, on page 189](#) or [Exploring Partition Choices with a Basic TSS, on page 185](#) for more about creating this file.) You must have this file if you are doing a quick partitioning run and have skipped pre-partitioning. You do not have to specify the file again if you have run pre-partitioning and specified it at that time.
- Specify partitioning constraints in a pcf file. See [Specifying PCF Constraints for Partitioning, on page 245](#).
- Optionally, specify synthesis attributes and constraints in an fdc file. Check FDC constraint syntax, using the report constraint\_check -syntax\_only command.
- Set other options as needed in a Tcl options file. Source the Tcl file.

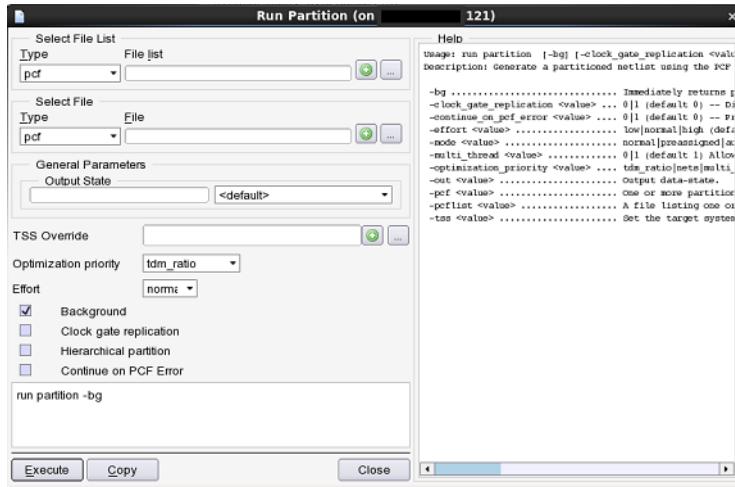
3. Optionally, set up more accurate area estimates for use during partitioning.

- Specify the run pre-partition -area\_est command if you want to generate more accurate area estimates. Specifying this argument generates a *design.est* file which you can use when you run partitioning. See [Working with Area Estimates for Partitioning, on page 337](#) for details.
  - Set cell\_utilization constraints in the pcf file to update area numbers for each instance.
4. Use the run partition command or select State->Run Partition in the GUI to partition the design.



- Set partitioning controls. See [Partitioning Controls, on page 332](#) for some typical controls.
- For better area estimation, use the controls described in [Working with Area Estimates for Partitioning, on page 337](#).
- For timing-aware optimizations, use the appropriate arguments to the run partition and run system\_route commands. See [Using Timing-Aware Partitioning, on page 341](#) and [Optimizing Multi-Hop Paths, on page 344](#) for details.
- Specify a simple TSS file with the -tss option and set -effort to low.
- Add constraints in a pcf file.
- Optionally, specify an fdc constraints file. Some constraints take effect at this stage.

To use the GUI to run the command, start with a pre-partitioned database, click Partition, and specify the command parameters. For descriptions of the dialog box options, see [Run Partition Dialog Box, on page 109](#) in the *Reference Manual*.



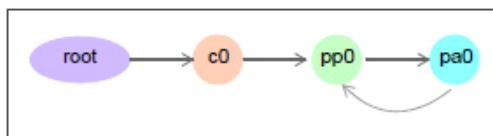
The tool automatically partitions the design and does global routing. After partitioning, the tool generates a multi-FPGA netlist. It also generates a new, partitioned database state. It generates an assignment pcf file and reports about the global routing.

## 5. Analyze results.

See [Analyzing Partition Result Files, on page 358](#) for information on interpreting results in

## 6. Make modifications to the TSS and pcf files as needed and rerun partitioning until you are satisfied with the results.

Use the output pcf from the previous run as input for the next run. Analyze the reports from each run and make changes based on the results. You can update the top level with results from the partitioned FPGAs, as described in [Updating the Top Level with FPGA Implementation Results, on page 435](#). You can also loop back to pre-partitioning and make changes at that stage, before running partition again.



The next step is to route the system level, as described in [Running System Route for the Top Level, on page 412](#). You can iterate between

partitioning and system routing, just as you do with multiple partitioning runs.

- To freeze partition results, export the assignments.pcf file and use it on the next run, as described in [Locking Down Partitions, on page 352](#).

## Partitioning Controls

The following table highlights some useful partitioning controls, but it is not an exhaustive list. See [Partition Constraint File Tcl Commands, on page 165](#) and [Target System Specification Commands, on page 243](#) in the *Command Reference Manual* for details about the available commands and constraints.

| Description                                      | Command/Constraint                                                                 |
|--------------------------------------------------|------------------------------------------------------------------------------------|
| run partition/run system_route Command Arguments |                                                                                    |
| Hardware specification                           | -tss                                                                               |
| PCF constraints                                  | -pcf<br>-pcf_continue_on_error                                                     |
| Timing-aware partitioning                        | -estimate_timing<br>-optimization_priority<br>-multi_hop_accuracy                  |
| Effort level                                     | -effort                                                                            |
| Area estimation                                  | -builtin_est<br>run pre-partition -area_est                                        |
| Clocks                                           | -clock_gate_replication<br>-clock_crossing_reduction                               |
| Feedthroughs                                     | -feedthrough_reduction<br>-illegal_feedthrough_reduction<br>-allow_feedthrough_tdm |
| PMUX decomposition                               | -pmux_decomposition                                                                |
| PCF File Constraints                             |                                                                                    |
| Resource utilization                             | bin_utilization                                                                    |
| Floorplanning                                    | assign_cell, cluster_cell, cluster_port                                            |
| Hierarchy control                                | cell_attribute -dissolve 0 1 auto, dissolve_control                                |
| Global routing controls                          | net_attribute                                                                      |

| Description                               | Command/Constraint                |
|-------------------------------------------|-----------------------------------|
| TSS File Commands                         |                                   |
| System configuration                      | board_system_create               |
| HAPS connections for top level            | board_system_configure -top_io    |
| Clock configuration                       | board_system_configure -clock     |
| Inter-FPGA connections<br>Daughter boards | board_system_create -interconnect |

## Partitioning Restrictions

The following restrictions apply when partitioning designs.

### General Partitioning Restrictions

- Rotate right (rotr) and rotate left (rotl) primitives inferred by the compiler cause the partitioner to error out. Work around this by using the netlist-based (srs) single-FPGA scripts generated after system generate, and ignore the HDL-based scripts. To generate netlist-based scripts, run system generate with the -database 1 argument.
- Enumerated data types are not supported.
- Support for member port types in composite data type records is limited. Only members with the following port types are supported for composite data type records: std\_logic, std\_logic\_vector, std\_ulogic, and std\_ulogic\_vector.
- Port types bit, bit\_vector, boolean, integer (signed and unsigned), and single- and multi-dimensional arrays are not supported as members in records. However, the partitioner can handle these types if they are not part of a record.

### Mixed-Language Restrictions

- You cannot use an entity with multiple architectures when Verilog is the language for the topmost level. This is because Verilog does not support multiple entity/architecture pairs. Here are some possible workarounds:
  - Run system generate with the -database 1 argument to generate netlist-based (srs) single-FPGA scripts for the partitions, and use them to run synthesis. Ignore the HDL-based scripts.

- Expose the mixed-project writer to only one entity/architecture pair. Do this by either not splitting views that contain entities with multiple views, or by generating wrappers for each entity/architecture pair with a unique entity name.
- If only a single architecture is used for prototyping, insert synthesis off and synthesis on pragmas around the unused architectures such that only one architecture is compiled for each entity.
- When entities with ports that are elements of records are at the top of the partitioned FPGAs, VHDL and mixed-language generated RTL code is not properly constructed. This results in a failure to compile mixed-language projects. If the entire record appears as a port, code generation is correct. You can avoid this by either using the srs netlist partitions, or by not placing entities with record elements at the ports of an FPGA boundary.
- When entities with ports of type character or string are at the top of the partitioned FPGAs, VHDL and mixed-language generated RTL code is not properly constructed and this results in a failure to compile mixed-language designs. Avoid this by either using the srs netlist partitions or by not placing entities with these types of ports at an FPGA boundary.
- With mixed-language designs, do not use reserved words in the other language as top-level FPGA port names, because this can cause linking errors during compilation. Either avoid using reserved words in port names or avoid mixed-language designs when reserved words are used as top-level FPGA ports.
- With mixed-language designs, using an entity with the same name as a module can cause the partitioner to error out when SLP HDL project writing is enabled. As a workaround, avoid using entities and modules with the same name or disable HDL SLP project writing.

## VHDL Restrictions

- Global VHDL signals (signal declarations in a package) are not supported in mixed-language partitions. Using these signal declarations in the RTL code restricts the tool to generating netlist-based partitions.
- With VHDL designs, entities that share the same name but which are located in different libraries and have logical differences can cause the partitioner to error out when Verilog partition writing is enabled. Either avoid using entities with the same name and different logic in different

libraries, or use srs-based single-FPGA scripts instead of HDL-based ones.

- When partitioning a VHDL design with signed division or remainder operations, the partitioner errors out when attempting to write out Verilog partitions. Because signed division or remainder operations in Verilog are not modeled in the same way as VHDL, you must disable the generation of Verilog partition output. Use mixed-language output when the source code must be used. For compiling and mapping the FPGAs, use the partition srs-based scripts instead of the HDL-based scripts.
- VHDL arrays are only supported with one-dimensional arrays of single-bit objects. Arrays of multi-bit objects (for example, `bit_vector`, `std_logic_vector`, and `integer`) are not supported.

### Verilog/System Verilog Restrictions

- With System Verilog designs, using interfaces or complex data types (structures or unions, for example) as a port can cause the partitioner to error out if it does not match the original port declaration. One possible workaround is to only use the netlist-based scripts for partitioning, and ignore the HDL-based scripts.
- If the top-level project module in a Verilog design has the same name as a sub-module but with a different letter case, you get an error when you simulate the entire system in VHDL, because the top (`vhp`) and a sub-entity have the same name in the VHDL name space. To avoid this error, do not name a top module with the same name as a sub-module if you have to consider VHDL case insensitivity, or do not simulate with VHDL.
- When compiling a Verilog partition, you must use the Verilog 2001 standard because parameters are passed formally. If your original project uses a Verilog 2001 reserved word as a port, net, or instance name, you get an “illegal use of reserved word” error. To avoid encountering the error, rename the object in the original source code.

## Partitioning Single-FPGA Designs

The typical prototyping flow with a single-FPGA target does not have a mechanism to specify hardware considerations, like daughter board interfaces or top-level port constraints. The hybrid flow described here shows you

how to specify hardware considerations for a single-FPGA design where the logic is confined to one FPGA, and includes hardware configuration information (TSS and PCF).

This is particularly useful for IP designers who no longer have to blend single-FPGA and multi-FPGA flows to accomplish what they need. It can also be used to set up designs that share the hardware system in multi-design mode (MDM).

1. Create a design database and compile it.

```
database create name -tech
run compile -srclist RTL_files
```

2. If you have set options, source the options file.

The options file is not stored with the database.

3. Prepare the design for partition.

- Create a TSS file with the system and daughter board definitions. For multi-user mode, make sure the TSS is compatible with other designs that share the hardware system.
- Include the hardware configuration command that designates a single FPGA (-single\_fpga\_target) in the TSS file:

```
board_system_configure -single_fpga_target fpgaName
```

This command assigns all the logic to the specified FPGA and locks all the other FPGA bins so no logic is assigned to them.

- Add timing constraints in an FDC file.
- Pre-partition the design, using the TSS and FDC file as inputs.

```
run pre_partition -tss tssFile -fdc fdcFile
```

4. Partition the design.

- If you have mixed systems, you can block systems for partitioning.
- Assign FPGA pins for the daughter boards in the PCF. Optionally, you can set cell utilization constraints.
- Generate partitions based on the PCF assignments.

```
run partition -pcf pcfFiles -area_est 0
```

- Check the partition log file for errors. The tool runs a DRC check on the constraints and reports errors in the log file.

```
@S2.3 |Single HAPS Target Flow DRC
@W: : | Disregarding soft cell assignment of cell 'myCell' to
bin(s) 'fb1.uB' due to single HAPS FPGA 'fb1.uA' target flow.
@E: : | Cell 'padder_' is replicated -hard to bin(s) 'fb1.uB'
fb1.uc' which is not supported for single HAPS FPGA 'fb1.uA' target
flow.
@E: : | Port 'out_304' is assigned -hard to bin(s)
'TOP_IO_HT3_mb1.B1' which is not supported for single HAPS FPGA
'fb1.uA' target flow.
Error in single HAPS FPGA target constraints.
```

5. Run system route to assign the pins:

```
run system_route -pcf pcfFiles
```

6. Create a single-FPGA database for synthesis:

```
run system_generate
```

The tool automatically turns off time budgeting when it generates the partitions.

## Working with Area Estimates for Partitioning

Area estimation runs during the pre-partitioning stage, but the estimates are used to guide partitioning when you use the run partition command. You can either use built-in, less accurate area estimation or more accurate area estimates that take longer to run. You can also reuse previously generated area estimates to save time.

- [Using More Accurate Area Estimates](#), on page 338
- [Reusing Area Estimation Data](#), on page 339
- [Bypassing Area Estimation to Save Runtime](#), on page 339
- [Using Built-in Area Estimates To Save Runtime](#), on page 340
- [Analyzing Area Estimation Results](#), on page 341

## Using More Accurate Area Estimates

For more accurate area estimation, you specify the `-area_est` argument when you pre-partition the design. This is instead of built-in area estimation which is a fast estimate that always runs during pre-partitioning, but which is less accurate ([Using Built-in Area Estimates To Save Runtime, on page 340](#)).

1. To use accurate area estimates from pre-partitioning, use this sequence of commands:

```
run pre_partition -area_est 1
export file design.est
run partition -builtin_est 0
```

- Run pre-partitioning with area estimation on. This generates a *design.est* file.
- Run the `export` list and `export file` *design.est* commands from the pre-partition stage to locate and export this file.
- When you run partitioning, specify the argument to use the area estimates from pre-partitioning instead of the built-in area estimation.

2. To use area estimates from a different pre-partition state, use these commands:

```
run pre_partition -area_est 1
export file design.est
database apply_state -import_est
run partition -builtin_est 0
```

- Run pre-partition and export the file as before.
- Import the area estimate into the current pre-partition database using the `database apply_state -import_est` command.
- Run partition with `-builtin_est 0`.

3. To use your own area estimates, use these commands:

```
database apply_state -import_est
run partition -builtin_est 0
```

- Generate the area estimate file. See [Reusing Area Estimation Data, on page 339](#) for additional information.
- Specify the area estimate file with the `database apply_state -import_est` *fileName* command.

- Run partition with -builtin\_est 0.
- Set cell\_utilization constraints in the pof file to update area numbers for each instance.

## Reusing Area Estimation Data

You can reuse area estimation data, either to save runtime or to use more accurate estimates, or to specify custom area estimates. Here are two common scenarios:

1. To reuse area estimates from a different pre-partition state: use a command sequence like the one below:

```
run pre_partition -out fileName
export file top.est
database set_state c0
run pre_partition -area_est 0 -out noest
database apply_state -import_est top.est
run partition ...
```

This sequence runs the normal area estimation flow during the first pre-partition run, and exports the associated area estimation .est file. Then, it disables area estimation for the second pre-partition run, and uses the estimates from the first pre-partitioning run for the partition stage.

2. To reuse area estimates generated by the Synopsys FPGA synthesis tools, use commands like the following:

```
run pre_partition -area_est 0
database apply_state -import_est <.file Synopsys FPGA synthesis>
run partition ...
```

These commands ensure that the pre-partition stage does not run area estimation and that the partition stage uses area estimation data (.est) previously generated in the FPGA synthesis tools.

## Bypassing Area Estimation to Save Runtime

To save runtime, you can either bypass area estimation or reuse previous area estimates.

1. To bypass area estimation when area estimation data is not available, use these commands:

```
run pre_partition -area_est 0
run partition ...
```

Setting -area\_est to 0 ensures that area estimation does not run during pre-partitioning, and that built-in area estimation is used to guide partitioning.

2. To bypass area estimation on an incremental run and reuse area estimates from a previous run, use this sequence of commands:

```
run pre_partition -out fileName
<make design/constraint changes>
database set_state c0
run pre_partition -area_est 0 -out fileName, from first pre-partition run
run partition ...
```

These commands ensure that the second pre-partitioning run does not re-run area estimation. The partition stage uses the area estimation data generated during the previous pre-partition run.

## Using Built-in Area Estimates To Save Runtime

Built-in area estimation is model-based, and is a very fast but less accurate method of area estimation. Built-in area estimation always runs during pre-partitioning, regardless of whether regular area estimation is ON or OFF. For more accurate area estimation, see [Using More Accurate Area Estimates, on page 338](#).

To save runtime, you can choose to use built-in area estimates, whether or not more accurate area estimation data is available:

1. To use built-in area estimates when regular area estimation is not available, use the following sequence of commands:

```
run pre_partition -area_est 0
run partition -builtin_est 1
```

2. To use built-in are estimates when normal area estimation is available, use these commands:

```
run pre_partition
run partition -builtin_est 1
```

These commands ensure that the partition stage uses built-in area estimates even though normal area estimation data is available.

## Analyzing Area Estimation Results

1. Run pre-partitioning.

The command generates an area estimate file called `design.est` and an srs netlist.

2. To view the estimate file, use report list and select the .est area estimate file.

You can export this file with the export command for use by other downstream database states, as described in preceding sections.

3. View the netlist with the view schematic command.

## Using Timing-Aware Partitioning

The tool provides different controls to drive timing-driven partitioning. The ones you choose to use depend on your design needs. Timing-aware partitioning is controlled by a combination of arguments to the run partition and run system\_route commands and options.

### Using `estimate_timing` with `optimization_priority`

The combination of the `-estimate_timing` and `-optimization_priority` arguments to the run partition and run system\_route commands determine the timing model used and let you tune your settings to suit your design needs. The following table summarizes how to use these options.

All setting combinations shown honor the TDM qualification mode you set (`tdm_control -qualification mode`), or default to all.

| Goal                                                     | Settings                                                  | Result                                                                                                                                                                                                                                                                     |
|----------------------------------------------------------|-----------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Best run time                                            | optimization_priority tdm_ratio<br>estimate_timing 0      | Optimizes the worst TDM ratio.                                                                                                                                                                                                                                             |
| Get slack values on inter-FPGA paths                     | optimization_priority tdm_ratio<br>estimate_timing 1      | Optimizes the worst TDM ratio. Runs timing estimation and adds slack values on inter-FPGA paths to the TD reports.                                                                                                                                                         |
| Best performance                                         | optimization_priority multi_hop_path<br>estimate_timing 1 | Runs timing estimation and does slack-based TDM insertion and multi-hop optimization.                                                                                                                                                                                      |
|                                                          | optimization_priority slack-mapped_timing_models 1        | Generates accurate timing models from mapped netlist to improve pre-tdm and post-tdm slack correlation between system route and system generate.<br>Improves correlation in multi-hop paths with clock groups, clock crossing paths and individual path timing exceptions. |
| Best runtime and performance (without clock constraints) | optimization priority multi_hop_path<br>estimate_timing 0 | Does TDM insertion based on board delays, optimizes the worst TDM ratio including multi-hop paths, and uses actual cable delays and 0 slack value for inter-FPGA paths.                                                                                                    |

For faster partitioning runs or to take clock constraints into account, use the feature described next, which is based on a timing graph model.

## Using Graph-Based Timing-Aware Partitioning

To achieve better partition results with a HAPS system, you can use top-down timing (TDT) which bases partitions on a timing graph model. Here, partitioning is based on timing graphs that the tool creates during pre-partitioning for each module, from the bottom up. During partition and system route, the tool composes a top-down timing graph based on the timing models, and

uses it to drive partitioning. Graph-based partitioning (or timing-aware partitioning, or TDT) only considers routing delays and assumes that all logic delays, except for signal delays that cross FPGAs, are 0.

Unlike `estimate_timing` model, the TDT does not require calculation time, so it is much faster. It also saves an additional cycle because it lets you change clock constraints and check the results at the system route state, without having to rerun and re-calculate, as you would with `estimate_timing`.

1. Set timing constraints.
  - Specify clocks and set constraints for them.
  - Set input and output delay constraints.

Timing-aware partitioning honors these constraints: `create_clock`, `set_clock_groups`, `set_false_path`, `set_multicycle_path`, `set_input_delay`, and `set_output_delay`. For information about these constraints, see [Specifying FDC Constraints, on page 20](#) in the *Compiler-Mapper Guide*.

There are some exceptions. Graph-based partitioning does not support the `set_max_delay` and `reset_path` constraints. It also does not support phase relationships defined with `create_clock`-waveform:

```
create_clock -name {clk} -period 10 -waveform {5.0 9.5} [get_ports {clk_in}]
```

Graph-based partitioning also does not support the propagation of two clocks when the constraint is applied on the same net, as shown here:

```
create_generated_clock -master_clock [get_clocks <clk_alias>] -add
```

2. Run pre-partitioning (run `pre_partition`) and partition (run `partition`)

The tool creates timing graphs for each module after pre-partitioning.

3. Specify that the timing graphs be used to drive routing with the run `system_route -optimization_priority slack` command.

This command enables routing based on top-down timing graphs, and inserts slack-based TDM. The partitioner loads the timing models for the current state and composes a complete timing graph for the design from the modules. Based on the timing graph, inter-FPGA TDM delay, and the fdc constraints (including reset path, false path, and multicycle path timing exceptions), it tries to minimize the worst slack of the design when it creates the partitions.

The router optimizes the design to minimize the worst case slack of the design using an estimate of the inter-FPGA TDM delay, top-down timing graph, and fdc constraints.

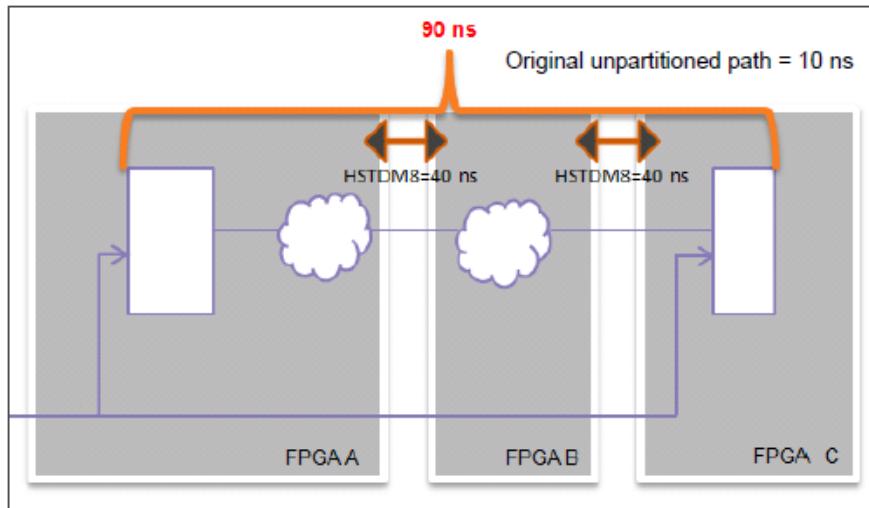
Use option `-mapped_timing_models` with `-optimization_priority slack` to generate accurate timing models from mapped netlist to improve pre-tdm and post-tdm slack correlation between system route and system generate.

4. Check the paths and timing reported in the critical path summaries in the log file.

## Optimizing Multi-Hop Paths

When logic for a path is partitioned between two FPGAs, a path from one FPGA to another FPGA is defined as a hop. A multi-hop path goes through multiple FPGAs: for example, from FPGA A to FPGA B to FPGA C. Multi-hop paths are important because they can significantly affect timing, based on the interconnectivity and TDM defined between FPGAs.

The following example shows a path that was 10 ns before partitioning, but which now takes 90 ns because of partitioning and the HSTDMD ratio defined.



The procedure below shows how to implement partitions that reduce the number of multi-hop paths. It is recommended that you use this mode as a final partitioning strategy after you have a final partition that you have qualified with `run system_route -optimization_priority -slack`.

1. When you partition the design, specify command arguments that minimize the number of multi-hop paths.

Set `-optimization_priority multi_hop_path` to enable hop optimization. This setting reduces the number of combinational paths that cross more than one FPGA.

`run partition -optimization_priority multi_hop_path`

- Optionally increase the effort spent on hop optimization to high:

`run partition -effort high`

This setting automatically sets the `-multi_hop_accuracy` option to 1, which uses a more accurate model for multi-hop path. This model takes longer to run, up to ten times longer than the estimation model (`-multi_hop_accuracy 0`). You can also specify the `-multi_hop_accuracy` option separately, if `-effort` is set to low or normal.

See [Multi-Hop Optimization Example, on page 347](#) for examples of optimization results.

After the partitioning run, the tool generates a log file that contains details about multi-hop paths and their effect on the design.

2. Check the following information in the multi-hop impact reports in the partition log:
  - TDM ratio to be used on multi-hop nets (Multi-Hop Ratio)
  - Total nets between FPGAs (Multi-Hop Usage + TDM Usage)
  - Remaining traces (Available TDM Traces - Multi-Hop Nets - Direct Nets)
  - TDM Ratio, which is determined by dividing the number of remaining nets, (not multi-hop or direct nets) by the number of remaining traces. For example, if there are 678 nets and 62 traces remaining, the TDM ratio is calculated as follows:

$$678 / 62 = 10.94 = \text{TDM Ratio}$$

The partition log file generated after partitioning contains reports about the impact of multi-hops in your design.

| Connectivity    | Direct Usage | Feedthrough Usage* | Multi-Hop Usage | Multi-Hop Ratio | TDM Usage | Available TDM Traces | TDM** Ratio |
|-----------------|--------------|--------------------|-----------------|-----------------|-----------|----------------------|-------------|
| mb1.uC[-]mb1.uD | 2^           | 0                  | 255             | 2               | 200       | 100                  | 5.33        |
| mb1.uB[-]mb1.uD | 4^           | 0                  | 309             | 2               | 659       | 330                  | 3.88        |
| mb1.uA[-]mb1.uD | 2            | 0                  | 248             | 1               | 678       | 312                  | 10.94       |
| mb1.uB[-]mb1.uC | 3^           | 0                  | 246             | 2               | 591       | 264                  | 4.25        |
| mb1.uA[-]mb1.uC | 1            | 0                  | 190             | 2               | 478       | 222                  | 3.79        |
| mb1.uA[-]mb1.uB | 4            | 0                  | 230             | 1               | 598       | 324                  | 6.64        |

3. Set options to use the lowest TDM ratio for multi-hop paths when you use run system\_route.

- Set -optimization\_priority multi\_hop\_path to enable hop optimization with the run system\_route command:

```
run system_route -optimization_priority multi_hop_path
```

- Set the lowest TDM ratio by enabling -estimate\_timing:

```
run system_route -estimate_timing 1
```

The tool uses inter-FPGA net slack to determine the TDM ratio. After run system\_route, it generates a timing estimate file called *design\_timeest.est*, which uses ACPM to estimate timing. It generates another report with slack and clock edge information called *tdm\_qualified\_timing.rpt*.

Other combinations of option settings have different effects on TDM. Refer to [Setting General Controls for TDM, on page 373](#) for details.

4. Check the following multi-hop information in the system router reports:

- Check the slack in the Routed Nets section of the System Router Report, which is a slack-based TDM report.

- Under the system-route database state, check the multi-hop paths in the Report of all nets qualified for TDM, which includes a multi-hop path report. This report shows details for each multi-hop path:

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> c0 (compile)   - Compile Log p0 (pre_partition)   - Area Estimation Log   - Netlist Editing and Optimization Log   - Pre_Map Log   - Target system specification log pa3 (partition)   - Partition Assignment Results   - Partition Log   - Partition Report   - Partitioned netlist creation log   - Report of all nets disqualified for TDM   - Report of all nets qualified for TDM     - Multi-hop Path Report   - Timing Estimation Log sg2 (system_generate)   - Pre_Map Log   - System generate log   - System Time Budget Log </pre> | <pre> Start points with multiple hops: 602 End points with multiple hops: 323 Printing a selected set of unique paths  Path #1 Hops      Bin/Ratio -----+   0 Port: TOP_10_BT3_mbl_C1 xxclr   0 Net:   mbl.uC   1 Net:   mbl.uA   2 Net:   mbl.uB           icore_sys.i_rst_b -----+ Path #2 Hops      Bin/Ratio -----+   0       mbl.uA   1 Net:   mbl.uB           icore_sys.ialb_mss_fab.i_buss_bvalid[0]   2 Net:   mbl.uB           icore_sys.ialb_mss_fab.i_buss_bvalid[1]   mbl.uB </pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

- To reduce runtime on subsequent runs, run system route with the timing estimate file generated on a previous run:

run system\_route -optimization\_priority multi\_hop -estimate\_timing 0  
{previousFile}

## Multi-Hop Optimization Example

The following figure shows the TDM ratios reported in the multi-hop impact report after partitioning with the default run partition command:

| Connectivity  | Direct | Feesthrough | Multi-Hop | Multi-Hop | TDM   | Available  | TDM <sup>**</sup> |
|---------------|--------|-------------|-----------|-----------|-------|------------|-------------------|
|               | Usage  | Usage       | Usage     | Ratio     | Usage | JDK Traces | Ratio             |
| mbl.uC mbl.uD | 2      | 0           | 855       | 2         | 280   | 180        | 5.33              |
| mbl.uS mbl.uD | 4      | 0           | 306       | 3         | 642   | 330        | 3.28              |
| mbl.uA mbl.uD | 2      | 0           | 848       | 1         | 678   | 312        | 10.64             |
| mbl.uB mbl.uC | 3      | 0           | 246       | 2         | 591   | 254        | 4.13              |
| mbl.uA mbl.uC | 1      | 0           | 196       | 2         | 478   | 222        | 3.75              |
| mbl.uA mbl.uB | 4      | 0           | 830       | 1         | 598   | 324        | 5.64              |

\* This report may not match the Global Route reports  
\*\* (Multi-Hop Traces Usage) = ((Multi-Hop Usage) + (Feesthrough Usage)) / (Multi-Hop Ratio)  
(TDM Ratio) = (TDM Usage) / (Available TDM Traces) = (Multi-Hop Trace Usage) / (Direct Usage)  
Note that in some cases the algorithm that calculates TDM ratio may deviate from this formula  
\* Some portion of Direct usage can uses non-measurable traces

The next figure shows the significantly improved TDM ratios reported in the Partitioner Ratio Estimate report after partitioning with run partition -optimization\_priority multi\_hop\_path:

| @s4.3.1 AP407   Partitioner Ratio Estimate Report |              |                    |                 |                 |           |                      |             |
|---------------------------------------------------|--------------|--------------------|-----------------|-----------------|-----------|----------------------|-------------|
| Connectivity                                      | Direct Usage | Feedthrough Usage* | Multi-Hop Usage | Multi-Hop Ratio | TDM Usage | Available TDM Traces | TDM** Ratio |
| mbl.uC[-]mbl.uD                                   | 2^           | 0                  | 295             | 2               | 280       | 180                  | 5.33        |
| mbl.uB[-]mbl.uD                                   | 4^           | 0                  | 309             | 2               | 669       | 330                  | 3.98        |
| mbl.uA[-]mbl.uD                                   | 2            | 0                  | 248             | 1               | 678       | 312                  | 10.94       |
| mbl.uB[-]mbl.uC                                   | 3^           | 0                  | 246             | 2               | 591       | 264                  | 4.25        |
| mbl.uA[-]mbl.uC                                   | 1            | 0                  | 190             | 2               | 478       | 222                  | 3.79        |
| mbl.uA[-]mbl.uB                                   | 4            | 0                  | 230             | 1               | 598       | 324                  | 6.64        |

Similarly, using -optimization\_priority multi\_hop\_path creates partitions that minimize the number of inter-FPGA nets. The inter-FPGA net count in the multi-hop path length report records this value. The following figure shows the difference when -optimization\_priority multi\_hop\_path is specified.

| Default                                      |                      |
|----------------------------------------------|----------------------|
| @s4.3.3 AP432   Multi-Hop Path Length Report |                      |
| Path Length                                  | Inter-FPGA Net_Count |
| 2                                            | 1374                 |
| 3                                            | 104                  |
| =====                                        |                      |
| @s4.4 AP432   Multi-Hop Path Length Report   |                      |
| Path Length                                  | Inter-FPGA Net_Count |
| 2                                            | 919                  |
| =====                                        |                      |
| With -optimization_priority multi_hop_path   |                      |

## Partitioning Clocks

The following procedures provide more information about handling clocks across partitions:

- [Handling Clock Skew by Replicating Clocks](#), on page 349
- [Replicating Clock Control Managers](#), on page 350
- [Rerunning Partition with Pre-Assigned Clocks](#), on page 351

## Handling Clock Skew by Replicating Clocks

If your design has clocks that cross FPGA boundaries, clock skew can degrade performance and introduce functional problems. One solution to this problem is to replicate the clock tree and insert clock buffers in the different FPGAs.

For the syntax of the commands mentioned in the following procedure, see [Partition Constraint File Tcl Commands, on page 165](#) in the *Reference Manual*.

1. Check the partitioner log for clocks and asynchronous resets that cross FPGA boundaries.

The report includes messages like the following:

```
@S4.5 AP406 | Partitioner Estimate of Clock & Asynchronous Reset Crossings
```

```
Cut clock: fb1.uD->{fb1.uc:} icore_sys.iarchipelago.iCCalb_cpu
_top_u_alb_cpu_clk_gated
```

2. Specify clock tree replication when you rerun partition:

```
run partition -clock_gate_replication 1
```

The software automatically replicates the clock tree in the FPGAs, and reports the results in the Automatic Clock Replication Summary section of the partition log, as shown in this example:

| Clock_Net                                                  | #Replicants | Target_Bins            | #Inputs | Prin |
|------------------------------------------------------------|-------------|------------------------|---------|------|
| *icore_sys.iarchipelago.iCCalb_cpu_top.u_alb_cpu.clk_gated | 4           | mbl.uC,mbl.uB,mbl.uA 1 |         | xcl_ |
| *N_2                                                       | 1           | mbl.uC,mbl.uB,mbl.uA 0 |         | xcl_ |
| TOTAL                                                      | 4           |                        |         |      |
| <hr/>                                                      |             |                        |         |      |
| *some replicants are counted multiple times                |             |                        |         |      |

It also generates a pcf file with the appropriate replication constraints, called `auto_replication.pcf`.

3. To manually replicate a cell to all bins, set this command in the pcf file:

```
replicate_cell {cellName} -all_bins 1
```

Use `clock_gate_replication_control` command to unlock bins that are otherwise locked, to allow clocks to be replicated. The bins are locked for all other operations

4. Use these pcf commands to disable replication:

- To disable manual clock replication, use the `replicate_cell` command:

```
replicate_cell {cellName} -all_bins 0
```

- To disable automatic replication and manual clock replication, use the `cell_attribute` command:

```
cell_attribute -auto_replicate 0
```

## Replicating Clock Control Managers

In general, it is recommended that you do not replicate clock managers like MMCMs (mixed-mode clock managers) on multiple FPGAs. The reason is that this could introduce phase differences between clocks, and can cause incorrect system-level timing. Replication might also result in hold violations and hardware failures. If design needs dictate the replication of MMCM logic on multiple FPGAs, you must do so manually, using PCF constraints.

Proceed with care when replicating CCMs (clock control managers).

1. Replicate the clock divider modules from the CCM.

- Instantiate the CCM in the top-level design and specify parameter values.
- Assign the clock module to an FPGA with a PCF constraint. For example:

```
assign_cell clock_control FB1.uA
```

- Replicate the clock module on other FPGAs, including the one to which it was assigned. This ensures that signals are synchronized. Use a PCF constraint, as in this example:

```
replicate_cell clock_control.cclk_gen_replica.cclkgen0 {FB1.uA FB1.uB}
```

For details about the PCF commands, refer to [Partition Constraint File Tcl Commands, on page 165](#) in the *Command Reference Manual*.

2. Assign the clock enable to the GCLK network with PCF constraints.

It is recommended that you connect CCM-generated clocks to the HAPS GCLK network and use the GCLK in all the FPGAs in your design.

```
assign_cell {clock_control.clk_cntr_en_keeper} {FB1.clk.bd_clk4}
net_attribute {clock_control.clk_cntr_en} -function GCLK -diffsingle -is_clock 1
net_attribute {clock_control.clk_cntr_en_int} -function GCLK
```

3. In the PCF, assign the trigger input port to gate the CCM, as in this example:

```
assign_port {trigger_in} -trace {FB1_A5_A[7]}
```

4. In the TSS file, define the GCLK to be used as the FPGA source. For example:

```
board_system_configure -clock {FB1.GCLK4} FPGA
```

For details about this command, see [board\\_system\\_configure, on page 245](#) in the *Command Reference Manual*.

For an example of MMCM replication, see SolvNet article 2345509. For specifics about replicating clock managers for debug with GSV, see [Instantiating and Configuring Clock Control Modules for GSV, on page 764](#).

## Rerunning Partition with Pre-Assigned Clocks

Use the technique described below when you are iterating, to remove cut clocks that were not resolved during initial partitioning. The technique reuses previous clock assignments, so you can build on existing results. It also saves runtime on subsequent partitioning runs.

1. Do an initial partitioning run with run partition, where clock replication assignments are made, as described in [Handling Clock Skew by Replicating Clocks, on page 349](#).
2. Export the constraints from this run with this command:

```
export file assignments.pcf
```

The assignments.pcf file is generated by the tool, and contains the clock constraints from the preceding run.

3. Reuse current clock assignments for the next run, with hard or soft constraints.
  - Do not edit the file if your intent is to resolve cut clocks. Leaving the constraints in the file as soft constraints (default) signals that the partitioner can optimize the clock assignments. Once the cut clocks have been resolved, you can make the constraints hard instead of soft.
  - To change the current constraints to hard constraints for subsequent runs, edit assignments.pcf. The file includes Tcl variables which are set

to soft by default. Change the value to hard. These are the Tcl variables:

```
set port_soft {-soft |-hard}
set cell_soft {-soft |-hard}
set repl_soft {-soft |-hard}
```

You can set each of these variables independently of the others.

- Use the variables to control assignments and replications, adding them to the commands in the assignments.pcf file. For example:

```
assign_port $port_soft port1 {portbin}
assign_cell $cell_soft cell1 {fb1.uA}
replicate_cell $repl_soft cell2 {fb1.uA fb1.uB}
```

The partitioner does not replicate clocks for cells that have hard constraints.

4. Rerun partition using assignments.pcf and the `-optimization_priority` preassigned argument.

```
run partition -optimization_priority preassigned -pcf assignments.pcf
 -clock_gate_replication 1 otherArgs
```

With preassigned set as an optimization priority, the partitioner uses the constraints set in assignments.pcf as a starting point. If you have unassigned cells, you get an error. The partitioner applies TDM qualification. This partitioner run is faster because it is incremental, and does not rerun certain partitioner operations.

## Locking Down Partitions

You can lock down partitions for incremental use or lock down the final solution.

### Locking Down Incremental Partitions

You can incrementally lock down partitions and save or freeze them so that you can make minor changes to them and reuse them on subsequent runs. You export a pcf file with final constraints to lock down partitions. The following steps demonstrate how to stabilize results for an incremental partitioning run.

1. Start with a database (partition or system route) where you are satisfied with the results.
2. Export the pcf file with the results.

The pcf file you export is different, depending on the state of your design.

- After partitioning, export assignments.pcf. For example:

```
database load db0
database set pa0
export file -list
export file assignments.pcf
```

Edit the assignments.pcf file to reflect minor incremental changes, without changing the RTL. The pcf file contains assignments to all the cells in the design, and can be used for predictable results. For example, if you want to add instrumentation to one FPGA but keep the others as is, delete the references to the FPGA you want to change, and use this updated file as input when you rerun partitioning.

- After system route, export router\_constraints.pcf. For example:

```
database load db0
database set sr0
export file -list
export file router_constraints.pcf
```

The router\_constraints.pcf file contains assignments to all the cells in the design, and can be used for predictable results. Make minor changes to this file if the partition changes, or if you need to move a net or cable to meet timing.

3. Specify the exported pcf file during the next run, using the run command appropriate to the design stage:

|           |                                    |
|-----------|------------------------------------|
| Partition | run partition -pcf assignments.pcf |
|-----------|------------------------------------|

|              |                                              |
|--------------|----------------------------------------------|
| System Route | run system_route -pcf router_constraints.pcf |
|--------------|----------------------------------------------|

4. To fix incremental trace assignments, use the net\_route pcf constraint as shown below:

- To control trace assignments to nets, use the -using\_trace option:

```
net_route {net1t} -from {FB1.uB} -to {FB1.uC}
 -using_trace {FB1_A12_A1[4]}
```

- To control how nets are grouped for TDM, use `-using_trace` with `-tdm_module`:

```
net_route {net1 net2 net3 net5} -from {FB1.uA} -to {FB1.uC}
 -using_trace {FB1_C12_C[6]} -tdm_module {HSTDMD_4by2}
 -tdm_net_name {tdm_net_1}
```

## Locking Down Final Partitions

When you are satisfied with the partition solution, lock down the solution by translating all abstract PCF constraints into TSS constraints.

1. Make sure the PCF does not contain any abstract definitions.
  - Make sure that all abstract definitions are translated into TSS constraints and that there are no abstract definitions in the PCF.
  - The PCF file should no longer contain any abstract constraints; it should only have assignment constraints.
2. Make sure the TSS fully defines the setup.

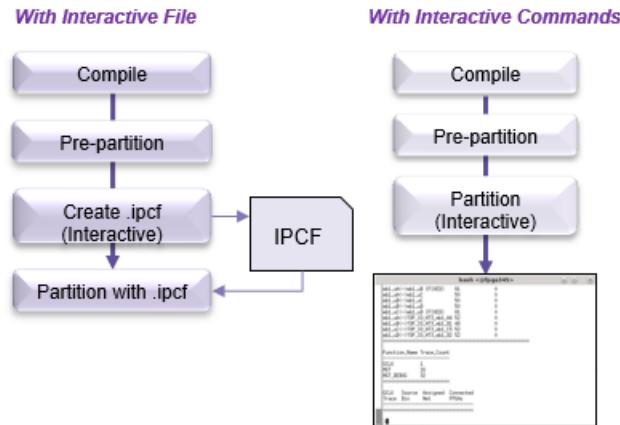
Translate all abstract PCF definitions into corresponding TSS definitions.
3. Run partition with these files.

The tool generates a partitioned database for system routing.

## Using Interactive Partitioning

Interactive partitioning is a way to access the partitioner and fine-tune partitioning by moving cells to different FPGAs. You can also get feedback on the impact of changes and reiterate.

You can run interactive partitioning through a file or run the commands individually in the shell window.



The procedure describes the steps to follow:

1. Set the license to enable the interactive partitioner: `-license_feature FpgalInternalBeta`
2. To run interactive partitioning through a file, do the following:
  - Create an interactive PCF file with the commands. See [Interactive Partitioning File, on page 357](#) for an example.
  - Specify the command with the `-ipcf` argument and the interactive PCF file:
 

```
run partition -interactive 1 [-ipcf filename]
```
3. To run an interactive partitioning session from the console, use this command:
 

```
run partition -interactive 1
```

When you specify the command without the file, an interactive shell window becomes available where you manually enter individual partitioning commands.

- Type `help` in the window to see a list of available commands at that stage. The list of available commands changes, according to the stage of interactive partitioning you are in.



The screenshot shows a terminal window titled "bash (on fpga1536mem1)". The window contains a list of commands, many of which are underlined, indicating they are clickable or have been recently used. The list includes:

- help
- hierarchical\_super\_bin
- net\_attribute
- net\_route
- partition\_optimization\_control
- pcf\_mode
- port\_attribute
- replicate\_cell
- report\_control
- reserve\_trace
- reset\_synchronize
- run\_flow
- run\_write\_results
- tdm\_control
- timing\_control
- trace\_group\_attribute
- tss\_list\_bins
- tss\_list\_functions
- tss\_list\_pins
- tss\_list\_properties
- tss\_list\_trace\_groups
- tss\_list\_traces

- Start an interactive partitioning session with `run_flow`. This command signals the switch to manual partitioning. If you type `help` again, you see a different set of available commands. Note that these interactive partitioning commands are slightly different from the regular PCF constraints, and that not all PCF choices are available for interactive partitioning. From this point up to the end of the session, you can only specify the interactive partitioning commands.

Refer to [Interactive Partitioning Commands, on page 239](#) in the *Command Reference* for command descriptions. For descriptions of individual commands, you can also type `commandName -help` in the window.

- Enter manual commands to make changes to partitions or get reports. For example, use these commands to get the worst paths and a report summary:

```
report_timing -type hops -worst_paths 10
report_timing -summary
```

- Run multiple iterations until you get the results you want. You can save and write out the results with the `solution_commit` and `run_write_results` commands.
- End an interactive session with this command: `exit`.

## Interactive Partitioning File

The commands in the file typically cover these phases:

1. Start manual partitioning with the `run_flow` command.
2. Analyze the design.
3. Assign cells and check results. Iterate through the analysis and assignment steps.
4. Save the solution and write out results (`solution_commit` and `run_write_results`).
5. End interactive partitioning with the `exit` command.

```
run_flow
report_timing -summary
report_timing -type haps -worst_paths 10
assign_cell F2.fbl.uA
assign_cell F3.fbl.uA
report_timing -summary
report_timing -type haps -worst_paths 10
solution_commit
run_write_results
exit
```

# Analyzing Partition Result Files

The following procedure provides general guidelines on what to check in the results file after one of the partitioning commands.

- [Checking the Target Specification \(TSS\)](#), on page 358
- [Methodology for Analyzing Partitioning at Different Stages](#), on page 360
- [Checking Partitioning Reports](#), on page 367
- [Parsing Partitioning Reports](#), on page 370

## Checking the Target Specification (TSS)

Check the target specification after a basic TSS file has been generated. You can do this after pre-partitioning or partitioning.

1. Specify the report `target_system -tss tssFile.tcl` command to generate a report.

The report file (report `pcf_tss_report.rpt`) contains detailed information about the run. It provides information on design considerations, like logic assignments, grouped objects, and dissolved cells. It also lists implied constraints, which are created as a side effect of other constraints. The report also shows the results of global routing for each external net, indicating nets with no legal routing with the `-unrouted` flag.

The file also provides information on hardware requirements, like the connection model, target system, the number of FPGAs required, and the requisite cabling. It lists the board bins and traces.

2. Run `view report target_system.rpt` to view the report.

Alternatively, list the generated reports with `report list` and export the file.

3. Check the report for the definitions of various bins:

|              |                                   |
|--------------|-----------------------------------|
| FPGA bin     | Logic elements of the target FPGA |
| Port bin     | Top-level port declarations       |
| External bin | External objects                  |
| Clock bin    | HAPS clock sources                |

```
@S1 AP245 |Target System Summary
=====
Name: 'ROUTE' Bins: 15
FPGA Bin: FB1.uA Pins 86 LUT 1243200 LUTM 621600 DFF 2486400 BRAM 1320 DSP 2160 IO 1200
FPGA Bin: FB1.uB Pins 87 LUT 1243200 LUTM 621600 DFF 2486400 BRAM 1320 DSP 2160 IO 1200
FPGA Bin: FB1.uC Pins 89-LUT 1243200 LUTM 621600 DFF 2486400 BRAM 1320 DSP 2160 IO 1200
FPGA Bin: FB1.uD Pins 89 LUT 1243200 LUTM 621600 DFF 2486400 BRAM 1320 DSP 2160 IO 1200
Port Bin: TOP_IO_HT3_1 Pins 50 PORT 50
Port Bin: TOP_IO_HT3_2 Pins 50 PORT 50
Port Bin: TOP_IO_HT3_3 Pins 50 PORT 50
Port Bin: TOP_IO_HT3_4 Pins 50 PORT 50
External Bin: FB1_rde Pins 105
External Bin: FB1.clk.bd_clk8 Pins 3
External Bin: FB1.clk.osc_100mhz Pins 2
External Bin: FB1.clk pll1 Pins 2
External Bin: FB1.clk pll2 Pins 2
External Bin: FB1.reset Pins 1
External Bin: umrbus_if Pins 0
```

4. You can also check the traces and identify clock drivers.

The report target\_system -tss command lists point-to-point traces and special functions like clock and reset networks. In the following example, the global clock bin FPB.clk pll1 drives the FB1.GCLK1 clock trace:

```
trace FB1.GCLK1 -function {GCLK hard} -from FB1.clk pll1 -pins {
```

It also indicates net splitting for multi-terminal trace assignments with the -split flag, and feedthroughs with the -feedthrough flag. It also shows the detailed trace assignments and clock routing.

5. Check trace information.

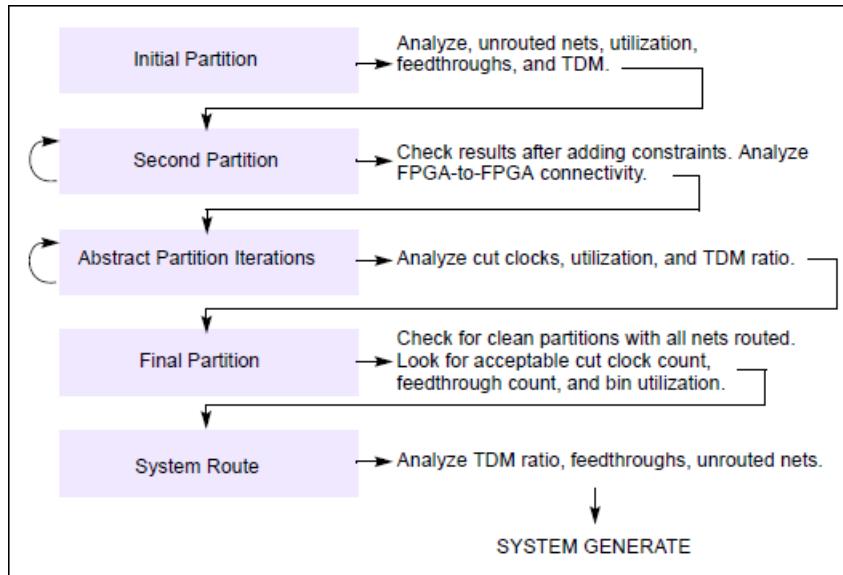
Go to the Traces->Traces Connecting Bins section, which shows connection information for daughter boards like this:

```
AP414 Section 2.2.2.1 TraceGroup T_DDR3.dimm_mb1.uA_1 Back
trace DDR3.DDR3_WE_B -connection_model DIRECT -group T_DDR3.dimm_mb1.uA_1 -pins {
 DDR3.dimm/WE_B
 mb1.uA/BC1.J5D[15]
}
trace DDR3.DDR3_S_B[1] -connection_model DIRECT -group T_DDR3.dimm_mb1.uA_1 -pins {
 DDR3.dimm/S_B[1]
 mb1.uA/BC1.J5D[6]
}
trace DDR3.DDR3_S_B[0] -connection_model DIRECT -group T_DDR3.dimm_mb1.uA_1 -pins {
 DDR3.dimm/S_B[0]
 mb1.uA/BC1.J5D[2]
}
trace DDR3.DDR3_RESET_B -connection_model DIRECT -group T_DDR3.dimm_mb1.uA_1 -pins [
 DDR3.dimm/RESET_B
 mb1.uA/BC1.J4C[9]
```

**Daughter Board Trace Name**

## Methodology for Analyzing Partitioning at Different Stages

The following figure shows five stages of partitioning, excluding pre-partitioning. The first four use the run partition command to generate partitioning solutions and fine-tune them, and the last uses the run system\_route command to implement the final solution.



This section describes how to leverage the partitioning reports in the context of the various phases shown above; for report-centered usage information, see [Checking Partitioning Reports, on page 367](#).

- [Analyzing Partition Information: Initial Partition Stage](#), on page 361
- [Analyzing Partition Information: Second Partition Stage](#), on page 362
- [Analyzing Partition Information: Initial Partition Stage](#), on page 361
- [Analyzing Partition Information: Final Partition Stage](#), on page 364
- [Analyzing Partition Information: System Route Stage](#), on page 365

## Analyzing Partition Information: Initial Partition Stage

1. Set up for the initial partition.
  - Pre-partition the design with `run pre_partition`.
  - Define connectivity using `abstract_tss -add_port_bin` commands to define cable connectivity.
  - To use port constraints to guide the solution, set `-constrain_connections` when you make port assignments in the PCF file.
2. It is recommended that you run partition with these arguments at the initial stage:

```
run partition -optimization_priority nets -hierarchy_dissolve_pass 1
```

These arguments reduce runtime.

For some designs, the `-effort high` argument can be very effective when used with `-optimization_priority nets`. However, the best options vary widely with the design, so experiment with the `-optimization priority` options for the best results.

3. Analyze nets in the partition log file.
  - Check whether partition goals can be achieved with the current parameters.
  - Check that constraints are applied correctly.
  - Check the Global Route Summary section for the number of unrouted nets. If unrouted nets are reported, check top-level port assignments, clock constraints, and cell assignment constraints.
  - If the total number of nets is high, it indicates possible partitioning problems because of the high trace utilization.
4. Check utilization values reported in the Bin Usage Summary section of the log file.

- Check that the resource utilization value reported for FPGA bins meets your goal values.
  - If you see higher utilization values at this stage, determine if your utilization goal is achievable. You can manually add up the utilization for all the bins and divide by the number of FPGAs to determine if setting additional constraints can improve the values or whether you have to adjust your utilization goals.
5. Check the Global Route Summary for feedthroughs and TDM.

At this point, just check and note if the value is feedthrough value is higher than the number of traces in an abstract channel at the maximum possible TDM ratio. If it is higher, this will have to be adjusted.

## Analyzing Partition Information: Second Partition Stage

The emphasis at this stage is to adjust physical constraints to get more realistic partition results.

1. Confirm that there are no unrouted nets.
2. Create a separate pcf file for each type of constraint listed below, or put them all in one pcf file:
  - Bin utilization constraints for bins, and a minimum LUT utilization constraint to balance utilization across bins. Example:

```
bin_utilization-all_bins-resource_ratio{0.65}
bin_utilization-min -all_bins-resource_ratio{LUT 0.30}
```

- Make high-level cell assignments to specific bins. For example:

```
assign_cell{sys} {fb_1 fb_2}
assign_cell{gpu} {fb_3 fb_4 fb_5}
```
  - Add dissolve constraints for specific cells. In general, do not dissolve cells that can fit into a bin at 65% utilization.
  - Add cluster constraints for specific cells, according to the design and cell connectivity.
3. Run partition with the new pcf constraints.
  4. Analyze results.
    - Check that the utilization constraints were honored.

- Check that there are no unrouted nets.
5. Analyze and adjust connectivity.

The goal is to create an abstract pcf file that uses less than 23 HT3 connectors for each FPGA.

- Check the Partitioner Ratio Estimate Report section of the partition log file for TDM ratio estimates. These estimates are based on partition engine estimates. Do not use the Global Route Ratio Report section, where the estimates are based on post-global route estimates.

Remember that the TDM ratio reported is single-ended. Double the value if you are using differential HSTDMD.

- Look for the next biggest TDM ratio available to physically account for the estimate. For example:

```
$S4.2.1 AP407 | Partitioner Ratio Estimate* Report
```

| Connectivity     | Available TDM Traces | Direct Usage | Feedthrough Usage* | TDM Usage | TDM** Ratio |
|------------------|----------------------|--------------|--------------------|-----------|-------------|
| fb_1.uA<->fb_.uD | 80                   | 5            | 0                  | 4507      | 60.09       |

For differential HSTDMD, you double the estimated TDM ratio of 60.09 to get 120.18. The next larger TDM ratio is 128:2.

- Use this data to modify the abstract pcf. Set constraints to lower the maximum ratio while making sure that the specified number of physical traces is available in the current system.

6. Create a new abstract pcf file based on the estimated TDM ratio results, and use it going forward.
- Change all cell assignments within this file from hard to soft by adding -soft. This change ensures that they are treated as guides instead of hard constraints.

| Hard Assignment                | Soft Assignment                      |
|--------------------------------|--------------------------------------|
| assign_cell {ufs_rx} {fb_1.uB} | assign_cell -soft {ufs_rx} {fb_1.uB} |

## Analyzing Partition Information: Abstract Partition Iteration Stage

The goal of this stage is to achieve a partition that satisfies initial results and a board file with some leeway for HT3 utilization. You might have to iterate through several times, checking reports and adjusting constraints. You might

even have to return to the previous phase and start over with a full abstract connectivity matrix but different cell constraints for placements, dissolving, and clustering.

1. Check for unrouted nets.

This might be caused by under-utilized channels between bins being removed. Fixes are design-specific. Some techniques are adding a new channel between bins, specifying cell placement, or replicating cells.

Check that there are no unrouted nets before proceeding.

2. Analyze and fix cut clocks in the Partitioner Estimate of Clock & Asynchronous Reset Crossings section of the log file.
3. Analyze utilization.
4. Analyze the TDM ratio results, as described in [Analyzing Partition Information: Second Partition Stage, on page 362](#).
5. Run and rerun partitioning until you have a satisfactory result.

These are the characteristics of a partition that is ready for final partitioning:

- It is fully routed
- It satisfactorily minimizes number of nets between FPGAs
- It has minimal cut clocks
- It has a low number of feedthrough nets
- It has a low number of multi-hop nets
- It has reasonable FPGA LUT and BRAM utilization, to make place and route faster and easier
- It utilizes an appropriate number of HT3 connectors for each FPGA

## Analyzing Partition Information: Final Partition Stage

The goal is to transfer the abstract connectivity to real cable assignments and create a database that is ready for the next phase, which is system routing.

1. Make sure the design is ready for the final partition, as described in [Analyzing Partition Information: Abstract Partition Iteration Stage, on page 363](#).

2. In the TSS file, use the auto connect feature (details) to specify only the widths of the FPGA-to-FPGA channels.

Note that when you use auto connect, the SLRs are not aligned. You can fix this later.

3. Run partition.
  - Partition with each FPGA utilizing HT3 connectors effectively  
Partition results are estimates only. Leave room for cable adjustments during the next stage (System Route) if needed.
  - Partition with desired maximum TDM ratio  
Partition results for TDM ratio are derived without considering timing. During system route, the tool assigns TDM ratio based on timing requirements. Allow a margin, for later adjustments during system route.
  - Use the `-optimization_priority multi_hop_path` argument to minimize multi-hop paths and improve timing.
4. Check the results.
  - There should be no unrouted nets.
  - There should be minimal nets between FPGAs
  - Cut clocks should be minimal.
  - There should be a minimal number of feedthroughs.
  - Bin utilization values should be satisfactory.

## Analyzing Partition Information: System Route Stage

1. Transfer abstract connectivity to a TSS file.

Check that there is some leeway for HT3 connector usage to modify cable connections if needed.

2. Add timing constraints in an fdc file.

The system route algorithm uses these constraints to optimize TDM ratios. It can apply lower TDM ratios on critical paths and relax the ratios on non-critical paths. However, this is still an estimate; the most accurate system-level timing results are reported after the system generate stage.

3. Run system route, specifying the files from the preceding two steps.

There are other useful options you can specify with the command:

- To use estimated timing numbers instead of the timing constraints, use the `-estimate_timing 1` option.
- Specify `-optimization_priority slack` to run timing estimation on paths that cross FPGA boundaries. Use this value as guidance. Currently, slack optimization does not take long net delays due to SLR crossings into account.
- To do SLR aware partitioning, run the `report slr_assignment` command from the partition node. If more than one cable is specified for a connection in the TSS file, the system route will choose the best cable based on the result of the SLR aware partitioning. The cell assignments based on the partitioning are saved in the `slr_cell_assignments.pcf` file. This file can be used to generate the partition results during the next iteration. The following is an example of an `slr_cell_assignments.pcf` file:

```
assign_cell $cell_soft {fb1.uA.rst_i} {fb1.uA.SLR0}
assign_cell $cell_soft {fb1.uA.u0} {fb1.uA.SLR0}
```

You can export the `slr_cell_assignments.pcf` file using the `export` command.

#### 4. Analyze the results.

- Check the Global Route Summary of the system route log file for information about the TDM ratio, feedthroughs, and unrouted nets.
- Check the Clock Summary section for information about the clock pairs (to/from clocks) that clock data on the FPGA interconnect and worst path data information before and after slack optimization.

#### 5. Analyze the slack to determine if the estimated numbers are enough to move forward.

- Check that the slack estimates are positive, regardless of the TDM ratio. Use Ctrl-f to search for “slack” in the log file. You can find slack information in the Clock Summary which reports the estimated slack on paths that cross FPGA boundaries.

If you have positive slack, you can move to the System Generate phase and confirm the estimates with system-level timing analysis.

- Analyze the slack estimation on critical paths. The High-Level System Performance Path Summary section provides details about the critical paths in the FPGA interconnect before and after slack

optimization. If there is a large negative slack, analyze the design and adjust constraints to improve timing. In some cases, you might have to go back to partitioning and redo the design. In many cases, adding an HT3 cable might be sufficient to lower the TDM ratio and fix the negative slack. If the TSS file allowed for extra cabling, you might not have to re-partition.

- Check the Multi Hops section of the log file for a summary of multi-hop paths, which can affect system clock frequency.

## Checking Partitioning Reports

The partitioner generates an `srr` log file and a `prf` report file after it finishes running. The log file contains high-level information about the run in chronological order, and includes error and warning messages. The `prf` report file contains detailed information about the assignments and is organized by function.

The files use a standardized format and syntax, which you can leverage to build your own parser and generate custom reports. See [Parsing Partitioning Reports, on page 370](#) for more information.

The following procedure describes what to check in a partitioning report; for information about leveraging information from partition reports generated at different stages, see [Methodology for Analyzing Partitioning at Different Stages, on page 360](#).

1. Check the log file for errors and warnings.

When constraints are applied to a collection of instances which are partitioned into separate FPGAs, you get a warning message. This is because constraints applied to a collection are assumed to apply only to the FPGA that contains the relevant logic and related path. As long as the logic related to a constraint is in a single partitioned FPGA, the constraints work without modification. When instances are in separate FPGAs, the constraints are only applied to the portion of the logic that is partitioned into that particular FPGA.

2. Check the status of TDM on asynchronous nets (AP255). The report displays whether the an asynchronous net is qualified for TDM or not. For example, for the net, R, the values under TDM are:
  - DT (Disabled Timing) - The nets are under different clock groups. This implies that the net is not qualified for TDM.

- ET (Enabled Timing) - The nets are under same clock groups.

| @S2.8 AP255   Asynchronous Set/Reset Net Information |     |             |
|------------------------------------------------------|-----|-------------|
| Net                                                  | TDM | Feedthrough |
| R                                                    | ET  | ET          |

3. For timing, check the following files:

- Check the partition.log file for information about multi-hop paths. Check the hop table and the hop path table. The tables list the connections between FPGAs. Check the Multi-hop usage column and multi-hop ratio to see if multi-hop paths require TDM, based on the number of traces available between FPGAs.
- Check timing.rpt for timing values.
- Check the multi-hop reports in the partition log for information about inter-FPGA timing. See [Optimizing Multi-Hop Paths, on page 344](#) for information about reducing the number of inter-FPGA hops.
- Check the partition log for information about clock skew and determine if clocks need to be replicated. See [Partitioning Clocks, on page 348](#) for details.

4. In the prf report file, check the Bin Usage section for bin utilization and the distribution of cells and ports in bins.

The following excerpt shows low utilization of FB1.uD, which can then be explored further.

```
fb.uA(0) Cells: 316 LUT 487466 DFF 334089 BRAM 1041 DSP 24 IO 4
fb.uB(1) Cells: 2562 LUT 715535 DFF 301760 BRAM 579 DSP 6
fb.uC(2) Cells: 3472 LUT 1154379 DFF 549655 BRAM 796
fb.uD(3) Cells: 6 LUT 402741 DFF 319405 BRAM 377
TOP_IO_HT3_1(4) Cells: 20 PORT 20
TOP_IO_HT3_2(5) Cells: 20 PORT 20
```

5. Check the Global Route Summary in the report file.

- Check TDM Ratio, and see if it meets your performance needs. TDM is time-domain multiplexing. See [Using HSTDMD and HSTDMD ERD, on page 376](#) for details about this technique.

@N: AP268 | Maximum TDM Ratio: 9

- Check for unrouted nets, and decide how to eliminate them.

```
@N: AP270 |Unrouted: 1
 FB1.clk.osc_100mhz -> FB1.uA 1
 Details:
 Unrouted net 'gclk8' from FB1.clk.bd_clk8 to FB1.uA - channel
 function mismatch
```

- Check for feedthrough routes and judge their effect on timing.

```
@N: AP268 |Feedthroughs: 0
```

- In the system route report, look for feedthrough errors reported on asynchronous set and reset nets:

```
@E: Feedthrough not allowed on async net
```

By default, the tool does not allow asynchronous set/reset nets to be used as feedthroughs to route signals through an intermediate FPGA when there are no available direct traces between the source and destination FPGAs. To override the default and use asynchronous sets/resets as feedthroughs, set a PCF feedthrough constraint:

```
net_attribute -feedthrough_allowed 1 netName
```

6. Check the Net Path Summary in the prf file for inter-FPGA trace requirements.

```
@S4.1 AP271 |Net Path Summary
@N: AP272 |4101 nets connected to bins FB1.uD FB1.uC with 500
 traces available
@N: AP272 |3223 nets connected to bins FB1.uC FB1.uA with 500
 traces available
@N: AP272 |3120 nets connected to bins FB1.uC FB1.uB with 500
 traces available
@N: AP272 |1411 nets connected to bins FB1.uD FB1.uB with 500
 traces available
@N: AP272 |710 nets connected to bins FB1.uB FB1.uA with 500
 traces available
```

Convert these requirements into firm trace constraints for the trace assignment stage ([Using Hierarchical Partitioning, on page 406](#)), after taking available connectors and cables and the effect of TDM into consideration. To get a list of available connectors, see [Identifying Connector Availability](#), next.

7. Check the Trace Usage section in the prf file to derive cabling requirements.

This example shows 10 connectors and cables required between FB1.uD and FB1.uB.

```
@S4.2 AP265 |Trace Usage
@N: AP266 |Trace _T_1419 (ix=25) Usage: 480/500 F: DEFAULT
@N: AP267 | FB1.uD->{FB1.uB} Net Usage 1370 (TDM 1370 DIRECT 0
 CLOCK 0) Async Ratio 3 Trace Usage 458
@N: AP267 | FB1.uB->{FB1.uD} Net Usage 42 (TDM 42 DIRECT 0
 CLOCK 0) Async Ratio 2 Trace Usage 22
```

If the total cabling and connector count exceed the resources, use one of these approaches to close the gap:

- Increase the TDM ratio while keeping the current connection scheme
- Use constraints to reduce interconnect
- Explore other connection schemes

## Identifying Connector Availability

To identify which connectors are free, do the following:

1. From the TCL shell, launch the TSS editor:  
`% launch tss`
2. From the TSS editor interface, run these commands to get the connector status:  
`% source <your TSS file>`  
`% board_system_list -connectors -available`
3. The software lists the connectors that are available for you to use. To find the connectors that are unavailable because they are being used by the system, use this command:  
`% board_system_list -connectors -occupied`

## Parsing Partitioning Reports

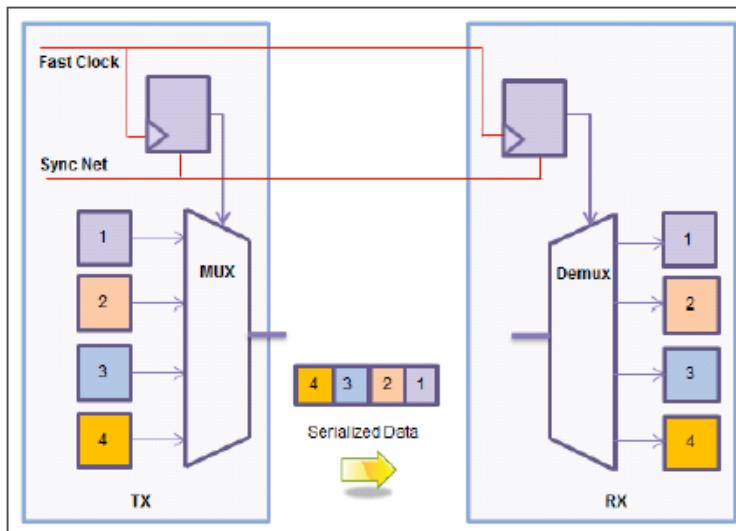
The partitioning log file and prf report files contain useful information about the partitioning run. You can build scripts that leverage the standard syntax in these reports and use them as pseudo commands to generate custom reports. Some of the commands are the same as those in the pcf constraints file.

The following table highlights some of the report syntax you can leverage. For details about the report syntax, refer to [Configuration Commands for Partitioning Reports, on page 173](#) in the *Reference Manual*.

| To check for ...                                                                           | Report Syntax                        |
|--------------------------------------------------------------------------------------------|--------------------------------------|
| Pin names, trace names                                                                     | target_system                        |
| Dissolved modules                                                                          | dissolve                             |
| Logic assignments: implied assignments, cell assignments, random logic assignments         | assign_cell                          |
| Port assignments                                                                           | assign_port                          |
| Unrouted nets (-unrouted), feedthroughs (-feedthrough), pin multiplexing ratios (-ratio)   | global_route                         |
| Net splitting (-split), feedthrough insertion (-feedthru)                                  | modify_net<br>(After system route)   |
| Clock routing assigned to global clock buffer port bins, connections to system PLLs (-pll) | assign_clock<br>(After system route) |
| Trace assignments                                                                          | assign_trace<br>(After system route) |

# Using Time Domain Multiplexing (TDM)

Partitioning an SoC design requires the interconnect signals of the SoC to be split across multiple FPGAs. You must take into consideration how to reconnect signals across FPGA boundaries and synchronize the FPGAs, so that the FPGAs function in the same way as they would in a single SoC. Typically the number of traces and pins required for the interconnect exceed the available I/Os on the FPGA. This results in interconnect congestion and bottlenecks. The solution to reducing congestion and the number of inter-FPGA I/O pins required, is to multiplex the pins.



The tool lets you specify time-domain multiplexing or TDM. This section describes these multiplexing models: HSTDMD (high-speed TDM), HSTDMD ERD (HSTDMD with error detection), and ACPM (asynchronous pin multiplexing). It covers the general TDM controls to set for all three models, as well as specific controls needed for each one. See the following sections for details:

- [Setting General Controls for TDM, on page 373](#)
- [Using HSTDMD and HSTDMD ERD, on page 376](#)
- [Working with HSTDMD Training Methods, on page 377](#)
- [Estimating System Frequency Based on HSTDMD Delays, on page 384](#)
- [Analyzing HSTDMD Results, on page 395](#)

- [Using Asynchronous Pin Multiplexing \(ACPM\), on page 400](#)
- [Using Multi-Gigabyte Transceivers for TDM, on page 401](#)

## Setting General Controls for TDM

You can specify various time-domain multiplexing controls in the pcf file. See [Partition Constraint File Tcl Commands, on page 165](#) in the *Command Reference Manual* for detailed descriptions of the commands mentioned here.

1. Set attributes for traces with the `trace_group_attribute` command in the pcf file.
2. In the pcf file, specify the TDM model with the `tdm_control -type` command.

You can set it to one of the following, with HSTDMD being the default:

- HSTDMD  
(High-speed time-domain multiplexing). This is the default mode. It uses source-synchronous differential signaling to eliminate the effects of skew and provide faster multiplexing speeds and increased pin capacity. It requires a HAPS board to be specified.
  - HSTDMD\_ERD  
(High speed time domain multiplexing with error detection). This mode is similar to HSTDMD, but the transmitter uses some of the bandwidth for sending checksum information with the user data. The receiver then separates and verifies the checksum information and, sets flag for checksum errors. The errors are subsequently processed by the HSTDMD monitoring feature in the debugger through the UMRBus. The HSTDMDxx\_ERD module type reflects the number of available data signals.
  - ACPM (Asynchronous pin multiplexing)  
This mode uses a shift-register based scheme for multiplexing.
  - DIRECT  
No multiplexing
3. Optionally, set TDM ratios with `tdm_control -min_ratio` and `tdm_control -max_ratio`. in the .pcf file.
    - Specify the `run system_generate` command, and check the slack in the `design_top_time_est.srr` file generated after the command has run.
    - Select a TDM ratio using the options described next. The tool sets a default ratio based on the model you selected, but you can use these

options to set specific limits. For more about TDM ratios, refer to [HSTDMD Ratios, on page 386](#).

For HSTDMD designs, you can estimate the system clock frequency by starting with the TDM ratio and adding the trace delay and user logic delay. These delays can vary greatly from design to design, depending on the number of partitions, amount of logic between flip-flops, and so on. See [Estimating System Frequency Based on HSTDMD Delays, on page 384](#) for details about TDM.

- Set maximum and minimum limits with these options:

```
tdm_control -min_ratio 8 -max_ratio 16
```

- If you want a specific ratio instead of a range, set the maximum and minimum to the same value.
- Set a higher TDM ratio for paths with higher positive slack and a lower ratio for paths with lower positive slack.

#### 4. Set the scope for TDM.

- If you did not specify a TDM ratio as described in the previous step, you can use the `tdm_control -allowed_modules` command to specify all the module types that are to be considered for TDM.
- Qualify nets for TDM with the `tdm_control -qualification_mode` command.

Analyze the slack based on the TDM ratio selected. The slack must be positive, both at the start and end points of the path. The best paths for qualification start and end at flip-flops. The next best choices either start or end at a flip-flop. Avoid feedthroughs. Typically you also exclude reset paths, clocks, critical paths with very high frequencies, and paths with negative slacks after TDM insertion.

- Override the general qualification by specifying the `net_attribute -tdm_qualified 0|1` command for individual nets or groups of nets. These settings on individual nets are always honored during the system route phase. |

#### 5. Set other model-specific controls.

See [Using HSTDMD and HSTDMD ERD, on page 376](#) and [Using Asynchronous Pin Multiplexing \(ACPM\), on page 400](#) for details.

#### 6. Calculate cabling requirements:

- Check the partitioning report for the number of traces required between FPGAs.

- Calculate one trace for each non-qualified net.
  - For qualified nets, the number of traces depends on the TDM ratio and signal direction. Remember to count two traces per channel for differential (diff) and one trace for single ended (SE).
7. To manually specify that TDM not be used between certain FPGAs, set the PCF trace\_group\_attribute to DIRECT.
- Use this when you need more control over net routing; for example when you do not want to use TDM for low slack nets. DIRECT changes the default assignment, and uses direct connections instead of TDM between the specified FPGAs.

8. Run partitioning and system route.

- See [Partitioning the Logic, on page 328](#) for details about the partition run.
- To influence the TDM results, specify the -optimization\_priority and -estimate\_timing options with the run system\_route command. For other information about running system route, see [Running System Route for the Top Level, on page 412](#). The following table summarizes system route options that affect TDM; the following option settings do not override any net\_attribute-tdm\_qualified settings on individual nets.

| Goal                                                            | Option Settings                                           | Effect on TDM                                                                                                                                                                                                                                                            |
|-----------------------------------------------------------------|-----------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Fast run time                                                   | optimization_priority tdm_ratio<br>estimate_timing 0      | Optimizes the worst TDM ratio. Honors the TDM qualification mode set with <code>tdm_control -qualification_mode</code> . The default is all.                                                                                                                             |
| Get slack values on inter-FPGA paths                            | optimization_priority tdm_ratio<br>estimate_timing 1      | Optimizes the worst TDM ratio. Estimates timing to generate the slack values on inter-FPGA paths and adds the information to the TDM qualification report. Honors the TDM qualification mode set with <code>tdm_control -qualification_mode</code> . The default is all. |
| Best performance                                                | optimization_priority multi_hop_path<br>estimate_timing 0 | Performs slack-based automatic TDM insertion and multi-hop optimization. Ignores the user-set TDM qualification mode.                                                                                                                                                    |
| Fast runtime and performance estimate without clock constraints | optimization_priority multi_hop_path<br>estimate_timing 1 | Performs TDM insertion based on board delays, and optimizes the worst TDM ratio including multi-hop paths. Uses actual cable delays and a slack value of 0 for inter-FPGA paths. Ignores the user-set TDM qualification mode.                                            |

## Using HSTDMD and HSTDMD ERD

HSTDMD is the default model for pin multiplexing. HSTDMD is a physically-aware, IOSERDES-based multiplexing model that uses synchronous differential signaling to achieve faster multiplexing speeds and greater pin capacity than ACPM. HSTDMD uses source-synchronous clocking for data, and synchronizes the data using circuit training. HSTDMD ERD is similar to HSTDMD, but includes error detection.

To use HSTDMD or HSTDMD ERD, do the following:

1. Set TDM controls like TDM ratios.

See [Setting General Controls for TDM, on page 373](#).

2. Set HSTDMD-specific controls, including HSTDMD training.

See [Working with HSTDMD Training Methods, on page 377](#).

3. Partition the design as usual.
4. Analyze results.

See [Analyzing HSTDMD Results, on page 395](#).

## Working with HSTDMD Training Methods

There are three methods for HSTDMD training, according to the HAPS system you are using:

HAPS-100 Proto\_rt is the default method for HAPS-100 and mixed system environment with HAPS-100 and HAPS-80 systems.  
[Using proto\\_rt for HSTDMD Training, on page 377](#)

HAPS-80 The HAPS-controlled method is the default.  
[Using HAPS-Controlled HSTDMD Training, on page 380](#)

HAPS-70 The reset hijack method is the default.  
[Using the Reset Hijack Method for HSTDMD Training, on page 382](#)

## Using proto\_rt for HSTDMD Training

Proto\_rt is a Confpro package used to run HAPS IP training, to initialize HAPS-100 systems, and to run MGTDM. The package can also be used to perform HAPS IP training in a mixed system environment where HAPS-100 and HAPS-80 systems are used. You can also use it in a HAPS-80 only environment, where IP training is integrated with Confpro.

See [Working with HSTDMD Training Methods, on page 377](#) for other methods of HSTDMD training to be used with other HAPS systems.

1. Set the HAPS\_INSTALL\_DIR to the ProtoCompiler version that you want to use for IP training.
2. Prepare the design.
  - Connect HAPS modules and make sure modules are cabled as defined in the TSS.

- Set the voltage on required connectors and program the required FPGAs on HAPS modules.
  - Make sure to close all HAPS modules connected to the host with the `cfg_close` command.
3. Create a hardware mapping file (HMF) file.

For details about creating this file, see [Creating an HMF File, on page 379](#).

4. Run `proto_rt` from the Confpro shell or from a Tcl file.

You can run the `proto_rt` package from the host machine where the HAPS systems are connected. Call the `proto_rt` package from the Confpro shell.

- Run `proto_rt` commands from the Confpro shell:

```
$HAPS_INSTALL_DIR/bin/confprosh
> proto_rt::run_ipinfra -hmf hmf_file.hmf -train all
```

Use the `-train all` option of the `proto_rt::run_ipinfra` command for training, and use IP-specific commands for debugging. This is the syntax for `proto_rt::run_ipinfra`:

```
proto_rt::run_ipinfra -hmf hmfJsonFile [-train all] [-report all]
[-report_verbose all] [-file filename] [-help]
```

Refer to [proto\\_rt::run\\_ipinfra, on page 96](#) in the *HAPS Prototyping Debugging Environment Reference* for descriptions of the arguments.

- Alternatively, save the `proto_rt` commands in a Tcl file, which you can then source in Confpro:

```
$HAPS_INSTALL_DIR/bin/confprosh run_commands.tcl
```

This is an example of the contents of the Tcl file:

```
package require proto_rt
proto_rt::run_ipinfra -hmf hmf_file.hmf -train all
```

5. Run HSTDMD training from a Confpro shell or a Tcl file, as described in the previous step.

```
proto_rt::run_ipinfra -hmf hmf_file.hmf -train all
```

If training fails, the command errors out.

6. Use these commands to generate a report to analyze the results and check for errors.

```
proto_rt:::run_ipinfra -hmf hmf.hmf -report all
proto_rt:::run_ipinfra -report_verbose all -hmf training.json
-file report.txt
```

The latter command generates a verbose report and saves it in the specified text file.

## Creating an HMF File

The following is an example for creating an hmf file. Enter the specified format in your *filename.hmf* file and modify to match your design.

1. Specify the complete device.

```
{
 "tsdmaphaps": {
 "FB1": {"serial": "X001234"},
 "FB2": {"serial": "X005678"}
 }
}
```

2. Specify individual FPGAs:

```
{
 "tsdmaphaps": {
 "FB1.uA": {"serial": "X001234", "fpga": "uA"},
 "FB2.uA": {"serial": "X005678", "fpga": "uA"}
 }
}
```

## HAPS-100 Bit Rate Table

The following table shows the bit rates for different lengths of cable (CON\_CABLE\_XX\_HT3) between FPGAs on HAPS-100 modules.

| Length XX<br>(cm) | SE/DIFF | Bit Rates |
|-------------------|---------|-----------|
| 25                | SE      | 1400      |
| 50                | SE      | 1400      |
| 100               | SE      | 1400      |
| 150               | SE      | 1000      |
| 200               | SE      | 1000      |
| 300               | DIFF    | 1000      |
| 400               | DIFF    | 1000      |

## Using HAPS-Controlled HSTDm Training

This is the default method for training HSTDm resets on HAPS-80 systems, but you can also use it with HAPS-70 systems. The advantages to using this system are that there are no changes to the reset tree, and no HSTDm reset requirements. In addition, there is no need to send the training signal to all the FPGAs as a CAPIM\_UI is used for cross-FPGA communication, and there is no dependence on cabling.

HAPS-80 systems can have single-ended (SE) HSTDm signals; HAPS-70 systems only support differential (diff) signals.

1. Set up the tool and environment, using the Confpro version included in the latest ProtoCompiler package.
2. Set general TDM settings, as described in [Setting General Controls for TDM, on page 373](#).
3. To override the default reset hijack method for HAPS-70 systems and use this method instead, specify this command in the pcf file:

```
tdm_control -hstdm_reset_trace {NONE}
```

For HAPS-80 systems, supervisor-controlled reset is the default setting. HAPS-80 systems do not have global resets, but the architecture includes local X\_USER\_RESET signals on each FPGA as part of the System IP. For design reset, use Confpro to control these X\_USER\_RESET signals.

4. Follow these trace assignment guidelines, and configure the FPGAs.
  - Connector J12 (HR bank) on HAPS-80 does not support HSTDMD capabilities.
  - Do not mix ratio 4 with higher ratios in the same cable or bank.
  - Use the bit rate recommended in the table under *Recommended Bit Rate Settings, on page 387*.
  - Configure the FPGAs independently, without using project.conf.
5. To run supervisor training, use Confpro.

When configured with Confpro, HAPS-controlled training runs automatically after the FPGAs are configured. The training uses the updated CAPIM\_UI and the UMRBus to control training. It broadcasts messages to all the CAPIMs, transmitting and then checking that the bit training and word training patterns were successfully started and received.

- You can also run on-demand training from the GUI or by using this Confpro command:

```
cfg_hstdm_train $cfg0 $hHstdm
```

You must use this method if you have configured the FPGAs independently, because HSTDMD training will not run automatically in this case.

- Validate the HSTDMD channels, using the `cfg_hstdm_selftest` command to run a self-check. With this command, the HSTDMD IP sends fixed known data to check the HSTDMD and cabling, reporting errors through the LEDs, and in a report that you can specify with the `-out` argument to the `cfg_hstdm_selftest` command.
- After running the self-check, reset the design to your data, because it will be set to the known data used for the test. Initiate the reset pulse for the design. Note that user reset and HSTDMD training are independent operations with supervisor-controlled training. You can issue a user reset before HSTDMD training is done, but the design will not be functional.

6. If training does not complete, do the following:
  - Use the Confpro `cfg_hstdm_report` command to check the status of the channels.

- Use the board bring up utilities to identify which channels are failing. You can also get this information by running `hstdm_report` in verbose mode.
  - Check the reports, as described in [Analyzing HSTDML Results, on page 395](#).
7. If there are errors during training, check for these typical causes:
- Check the connectors for bent pins.
  - Check that the cable is properly seated. Re-seat the HT3 cable, or replace it.
  - Check the verbose version of the board bring-up utility for information.

## Using the Reset Hijack Method for HSTDML Training

The reset hijack method is the default for HAPS-70 systems. For more information about reset hijacking, refer to [HSTDML Training with the Reset Hijack Method, on page 383](#).

1. Set general TDM settings, as described in [Setting General Controls for TDM, on page 373](#).
2. Set the `tdm_control -hstdm_reset_trace` command in the `.pcf` file to identify the reset signal to use as the global reset trace.

`tdm_control -hstdm_reset_trace {traceName}`

- If you have multiple, top-level design resets, select one to act as the master reset. This signal is then multiplexed using HSTDML.
- Do not connect the reset to multiple FPGAs, as this can cause an error if the design cannot tolerate different reset release times at each FPGA. The recommended practice is to synchronize the design reset in one FPGA and then distribute this reset to the other FPGAs using local, on-board traces. If your top-level reset is connected to more than one FPGA, you get a warning message like this one:

```
@W|Multiple chips use the syn_hstdm_reset_pin (global reset)
```

The specified global reset is hijacked and held active until the HSTDML training sequence is complete.

This signal holds the design in a reset state until circuit training is complete. Generally, the global reset net is the `RESETn` trace on the HAPS board. For the HAPS-80 default methodology, see [Using HAPS-Controlled HSTDMD Training, on page 380](#).

- To assign a global reset as the default, use syntax like this:

```
assign_port {rst} {fb1.reset}
net_attribute {rst} -function {RESET}
tdm_control -type HSTDMD -max_ratio 8 -hstdmd_reset_trace fb1.RESETn
```

In this example, `RESETn` is the global reset trace name; use `net_attribute` to assign it for circuit training. `fb1` is the board name from the TSS `board_system_create -add` command:

```
board_system_create -add HAPS-70-S24 -name fb1
```

- To assign a non-global signal as the default, use syntax as shown below:

```
assign_port {rst} -trace {fb1_A1_A[10]}
tdm_control -type HSTDMD -max_ratio 8 -hstdmd_reset_trace {fb1_A1_A[10]}
```

This example sets the non-global signal `fb1_A1_A[10]` as the global reset trace. As with the previous example, `fb1` is the board name defined with the TSS `board_system_create -add` command. Specify the appropriate connector with `-hstdmd_reset_trace`: `fb1_A1_A[10]`.

## HSTDMD Training with the Reset Hijack Method

HSTDMD uses a programmable delay element on the receiving side of the HSTDMD data. Before transmitting design data on an HSTDMD channel, it first transmits a known data pattern to determine the range of programmable delay. The programmable delay element is then set to the mid-value of the range. The process of setting this HSTDMD delay is termed *training*.

- Training is on a per HSTDMD channel basis, and is conducted independently in each FPGA. This is common to both the reset hijack and HAPS-controlled training methods.
- Training occurs after power up and after system reset is released.
- The reset signal is *hijacked* by disconnecting the user logic and inserting training logic. The user logic is connected to the new reset generated by HSTDMD. When the global reset is asserted (for example, while program-

ming the FPGAs) and during training, the user design is held in a reset state.

- Until all HSTDMD channels in all FPGAs are trained, the hijacked reset is asserted to hold the design in the reset state.
- A single root FPGA collects all the *training done* signals from all FPGAs on the board and then releases the reset for the design with the *reset\_out* signal. All other FPGAs that use HSTDMD are defined as intermediate FPGAs. They forward the *reset\_in* signal to the downstream FPGA as *reset\_out*, use the *reset\_in* signal for logic within the FPGA, and forward the *training\_out* signal upstream to the root FPGA after combining it with their local training status.

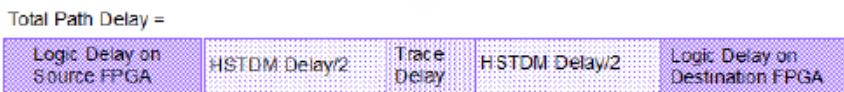
## Estimating System Frequency Based on HSTDMD Delays

Estimating system frequency requires that you estimate total path delays. Delay calculations for sequential-to-sequential paths with TDM must take trace delay into account, in addition to the user logic delay and HSTDMD delay.

The following procedure and subsequent tables illustrate how to compute the total delay of paths with HSTDMD in order to estimate system frequency.

1. When estimating system frequency, make sure that the system period is greater than the total delay of the path with HSTDMD.

The following figure shows the components of delay on a path with HSTDMD (not to scale):



2. Use realistic estimates for the user logic delay.

The data path delay on user logic is heavily dependent on the eventual placement and routing solution. When calculating the clock period, it is recommended that you leave enough margin to compensate for the potential extra delay introduced by place-and-route variances.

3. Estimate the HSTDMD ratio and bit rate.

See the following for more information:

HSTDMD ratio [HSTDMD Ratios, on page 386](#)

Bit rate [Recommended Bit Rate Settings, on page 387](#)

For mixed system configurations, use the recommended bit rate that it is supported by both systems. For example, if you have a mixed configuration of HAPS-80 and HAPS-70 systems, use the recommended differential bit rate from the HAPS-70 table, not the HAPS-80 single-ended bit rate.

4. Find the sum of the HSTDMD delay and trace delay.
  - HSTDMD delay is based on the ratio and bit rate. Consult the appropriate table for your HAPS system, and use the delay listed in the table for that ratio and bit rate:

[HAPS-80 HSTDMD Delay Table, on page 391](#)

[HAPS-70 and HAPS-DX7 HSTDMD Delay Table, on page 393](#)

[HAPS-80 HSTDMD\\_ERD Delay Table, on page 392](#)

[HSTDMD\\_ERD Delay \(HAPS-70 and HAPS-DX7\), on page 394](#)

- Add 5-8 ns to the HSTDMD delay in the table to account for trace delay. See [FPGA-to-FPGA Trace Delay Table, on page 394](#) for more information.

Trace delay is the inter-FPGA pin-to-pin delay and does not include the input and output buffer delays. The buffer delays are included in the HSTDMD delay.

## Constraining HSTDMD Timing Paths

There are three kinds of HSTDMD paths: user logic to user logic; intra-HSTDMD paths, and paths between HSTDMD and user logic (to/from). The most critical HSTDMD timing paths are usually the paths between the HSTDMD and the user logic.

The following procedure outlines the steps needed to set up HSTDMD timing paths.

1. Run time budgeting to create constraints between the user clock and the HSTDMD clock.

The user clock and the HSTDMD clock are asynchronous, so define timing exception constraints for them before the design goes through synthesis, place, and route. The tool creates DPO (data path only) constraints, which are like maximum delay constraints but are independent of clock delay.

2. Check these `set_datapath_only` constraints for negative values.

The paths have negative values when the period is less than the TDM delay. Negative values affect place and route quality and run time. Try changing the clock period to a more reasonable value, and rerun the design stage.

3. Check timing closure on intra-HSTDMD timing paths.

There are two primary clocks in the HSTDMD IP: a 550 MHz bit clock used by IOSERDES and a 137.5 MHz word clock used by the HSTDMD logic to clock the user interface. The bit clock is sent to the receiver logic on the FPGA as a source-synchronous (SS) clock. The tool automatically ensures word clock timing.

## HSTDMD Ratios

These are some guidelines to follow when setting HSTDMD ratios:

- All paths are not critical, so they do not all need to be fast; for example, false paths, asynchronous clock crossings, slow clocks, and debug paths.
- You can mix HSTDMD ratios. The source sync clock is shared across ratios.
- Match TDM ratios to the slack on the path. Use lower ratios on critical paths and higher ratios on non-critical paths.
- Free up traces by using higher ratios on non-critical paths.
- An HSTDMD ratio multiplexes the number of nets specified in the ratio, and each original signal needs a constraint. Each net or bit requires its own DPO constraint, and each clock edge needs a different DPO constraint.
- The default bit rate of 1400 Mbps does not support a ratio of 4; this bit rate only supports ratios of 8 and above. To use a ratio of 4, change the bit rate to 1200.

- HAPS-80 systems use single-ended (SE) HSTDMD. Instead of the two traces used by differential HSTDMD, single-ended HSTDMD uses only one trace. For the same ratio, the delay values are the same for single-ended and differential HSTDMD. However with single-ended HSTDMD, you can use lower ratios for the same number of cables, because single-ended HSTDMD doubles the band-width and thus effectively reduces the TDM ratio by half.

For example for a bit rate of 1400, if you have a ratio of 64 with a delay of 92 ns in differential mode, you can assume a ratio of 32 with a delay of 70 ns in single-ended mode.

## Recommended Bit Rate Settings

Recommended bit rate settings for different configurations vary, depending on the HAPS system configuration and cable type. When using a global bit rate for chained HAPS systems that are part of the same design, the recommended setting is the lowest of the recommended bit rates, even if that cable is not used for HSTDMD.

You can only use the listed short and long cables for HSTDMD. No other cables or extension boards (e.g EXT\_CABLE\_40\_HT3 or RISER\_HT3) are supported for HSTDMD.

| Cable             | HAPS-DX7      | HAPS-70<br>Mixed HAPS-80<br>and HAPS-70 | Bit Rate<br>(Mbps)                                           |
|-------------------|---------------|-----------------------------------------|--------------------------------------------------------------|
| CON_CABLE_100_HT3 | 800<br>(Diff) | 1100<br>(Diff)                          | 1400<br>(SE, ratio 8 and<br>higher)<br>1200<br>(SE, ratio 4) |
| CON_CABLE_150_HT3 | -             | 800<br>(Diff)                           | 1000<br>(SE)                                                 |

| <b>Cable</b>      | <b>HAPS-DX7</b> | <b>Bit Rate<br/>(Mbps)</b>                       |                |
|-------------------|-----------------|--------------------------------------------------|----------------|
|                   |                 | <b>HAPS-70<br/>Mixed HAPS-80<br/>and HAPS-70</b> | <b>HAPS-80</b> |
| CON_CABLE_200_HT3 | -               | 800<br>(Diff)                                    | 1000<br>(SE)   |
| CON_CABLE_300_HT3 | -               | -                                                | 1000<br>(Diff) |
| CON_CABLE_400_HT3 | -               | -                                                | 1000<br>(Diff) |

To enable bit rates based on cable type set the PCF control `tdm_control -hstdm_auto_bitrate_selection` to 1. The `system_route.log` will show that HSTDMD Bitrate is set to auto and HSTDMD Bitrate per Cable as on.

## HSTDMD Delay Tables

HSTDMD delay is primarily a function of ratio and bit rate. HSTDMD delay on the HAPS-80 system differs from delay on systems from the HAPS-70 series. The following tables show the delays for different systems based on different ratios and bit rates.

- [HAPS-100 HSTDMD Delay Table](#), on page 388
- [HAPS-100 HSTDMD Delay Table](#), on page 388
- [HAPS-80 HSTDMD Delay Table](#), on page 391
- [HAPS-80 HSTDMD\\_ERD Delay Table](#), on page 392
- [HAPS-70 and HAPS-DX7 HSTDMD Delay Table](#), on page 393
- [HSTDMD\\_ERD Delay \(HAPS-70 and HAPS-DX7\)](#), on page 394

The values are subject to change with the release, depending on ongoing enhancements. HSTDMD support ratios up to 256, while HSTDMD ERD supports up to 240.

### HAPS-100 HSTDMD Delay Table

The table shows HSTDMD delay values in ns, excluding trace delay:

| Ratio | Bit Rate  |           |           |
|-------|-----------|-----------|-----------|
|       | 1400 Mbps | 1200 Mbps | 1000 Mbps |
| 8     | 28        | 31        | 36        |
| 16    | 37        | 43        | 50        |
| 24    | 43        | 49        | 58        |
| 32    | 49        | 56        | 66        |
| 40    | 54        | 63        | 74        |
| 48    | 60        | 69        | 82        |
| 56    | 66        | 76        | 90        |
| 64    | 71        | 83        | 98        |
| 72    | 77        | 89        | 106       |
| 80    | 83        | 96        | 114       |
| 88    | 89        | 103       | 122       |
| 96    | 94        | 109       | 130       |
| 104   | 100       | 116       | 138       |
| 112   | 106       | 123       | 146       |
| 120   | 111       | 129       | 154       |
| 128   | 117       | 136       | 162       |
| 160   | 146       | 169       | 202       |
| 192   | 169       | 196       | 234       |
| 224   | 191       | 223       | 266       |
| 256   | 214       | 249       | 298       |

### HAPS-100 HSTDMD\_ERD Delay Table

The table shows HSTDMD \_delay values in ns, excluding trace delay:

| Ratio | Bit Rate  |           |           |
|-------|-----------|-----------|-----------|
|       | 1400 Mbps | 1200 Mbps | 1000 Mbps |
| 7     | 31        | 36        | 42        |
| 15    | 37        | 43        | 50        |
| 22    | 43        | 49        | 58        |
| 30    | 49        | 56        | 66        |
| 37    | 54        | 63        | 74        |
| 45    | 60        | 69        | 82        |
| 52    | 66        | 76        | 90        |
| 60    | 71        | 83        | 98        |
| 67    | 77        | 89        | 106       |
| 75    | 83        | 96        | 114       |
| 82    | 89        | 103       | 122       |
| 90    | 94        | 109       | 130       |
| 97    | 100       | 116       | 138       |
| 105   | 106       | 123       | 146       |
| 112   | 111       | 129       | 154       |
| 120   | 117       | 136       | 162       |
| 150   | 146       | 169       | 202       |
| 180   | 169       | 196       | 234       |
| 210   | 191       | 223       | 266       |
| 240   | 214       | 249       | 298       |

## HAPS-80 HSTDMDelay Table

The table shows HSTDMDelay values in ns, excluding trace delay.

| Ratio | Bit Rate  |           |           |           |          |
|-------|-----------|-----------|-----------|-----------|----------|
|       | 1400 Mbps | 1200 Mbps | 1100 Mbps | 1000 Mbps | 800 Mbps |
| 4     | -         | 38        | 40        | 44        | 77       |
| 8     | 50        | 56        | 60        | 66        | 80       |
| 16    | 58        | 68        | 72        | 80        | 96       |
| 24    | 64        | 74        | 80        | 88        | 106      |
| 32    | 70        | 80        | 88        | 96        | 116      |
| 40    | 76        | 88        | 94        | 104       | 126      |
| 48    | 82        | 94        | 102       | 112       | 136      |
| 56    | 88        | 100       | 108       | 120       | 146      |
| 64    | 92        | 108       | 116       | 128       | 156      |
| 72    | 98        | 114       | 124       | 136       | 166      |
| 80    | 104       | 120       | 130       | 144       | 176      |
| 88    | 110       | 128       | 138       | 152       | 186      |
| 96    | 116       | 134       | 146       | 160       | 196      |
| 104   | 122       | 140       | 152       | 168       | 206      |
| 112   | 128       | 148       | 160       | 176       | 216      |
| 120   | 132       | 154       | 168       | 184       | 226      |
| 128   | 138       | 160       | 174       | 192       | 236      |
| 160   | 168       | 194       | 212       | 232       | 288      |
| 192   | 190       | 220       | 140       | 264       | 328      |
| 224   | 214       | 248       | 270       | 296       | 268      |
| 256   | 236       | 274       | 298       | 328       | 408      |

It might be difficult to get timing closure using ratio 4 at 1400 with speed grade -1. In this case, try the following techniques:

- Run Vivado with other options to get a different solution.
- Prevent the tool from using ratio 4 by setting tdm\_control -min\_ratio to 8 in the PCF file.
- Use a lower bit rate, like 1200.
- Use speed grade -2.

## HAPS-80 HSTDMD\_ERD Delay Table

The table shows HSTDMD\_ERD delay values in ns, excluding trace delay.

| Ratio | Bit Rate  |           |           |           |          |
|-------|-----------|-----------|-----------|-----------|----------|
|       | 1400 Mbps | 1200 Mbps | 1100 Mbps | 1000 Mbps | 800 Mbps |
| 7     | 52        | 60        | 66        | 72        | 86       |
| 15    | 58        | 68        | 72        | 80        | 96       |
| 22    | 64        | 74        | 80        | 88        | 106      |
| 30    | 70        | 80        | 88        | 96        | 116      |
| 37    | 76        | 88        | 94        | 104       | 126      |
| 45    | 82        | 94        | 102       | 112       | 136      |
| 52    | 88        | 100       | 108       | 120       | 146      |
| 60    | 92        | 108       | 116       | 128       | 156      |
| 67    | 98        | 114       | 124       | 136       | 166      |
| 75    | 104       | 120       | 130       | 144       | 176      |
| 82    | 110       | 128       | 138       | 152       | 186      |
| 90    | 116       | 134       | 146       | 160       | 196      |
| 97    | 122       | 140       | 152       | 168       | 206      |
| 105   | 128       | 148       | 160       | 176       | 216      |
| 112   | 132       | 154       | 168       | 184       | 226      |
| 120   | 138       | 160       | 174       | 192       | 236      |
| 150   | 168       | 194       | 212       | 232       | 288      |

| Ratio | Bit Rate  |           |           |           |          |
|-------|-----------|-----------|-----------|-----------|----------|
|       | 1400 Mbps | 1200 Mbps | 1100 Mbps | 1000 Mbps | 800 Mbps |
| 180   | 190       | 220       | 240       | 164       | 328      |
| 210   | 214       | 248       | 270       | 196       | 368      |
| 240   | 236       | 274       | 298       | 328       | 408      |

### HAPS-70 and HAPS-DX7 HSTDm Delay Table

The table shows HSTDm delay values in ns, excluding trace delay.

| Ratio | Bit Rate  |           |         |
|-------|-----------|-----------|---------|
|       | 1100 Mbps | 1000 Mbps | 800Mbps |
| 4     | 26        | 26        | 30      |
| 8     | 40        | 42        | 50      |
| 16    | 54        | 58        | 72      |
| 24    | 62        | 66        | 82      |
| 32    | 72        | 78        | 98      |
| 40    | 80        | 86        | 108     |
| 48    | 88        | 94        | 118     |
| 56    | 94        | 102       | 128     |
| 64    | 102       | 110       | 138     |
| 72    | 110       | 118       | 148     |
| 80    | 116       | 126       | 158     |
| 88    | 124       | 134       | 168     |
| 96    | 132       | 142       | 178     |
| 104   | 138       | 150       | 188     |
| 112   | 146       | 158       | 198     |
| 120   | 152       | 166       | 208     |
| 128   | 160       | 174       | 218     |

## HSTDMD\_ERD Delay (HAPS-70 and HAPS-DX7)

The table shows HSTDMD\_ERD delay values in ns, excluding trace delay.

| Ratio | Bit Rate  |           |          |
|-------|-----------|-----------|----------|
|       | 1100 Mbps | 1000 Mbps | 800 Mbps |
| 7     | 44        | 46        | 58       |
| 15    | 54        | 58        | 72       |
| 22    | 62        | 66        | 82       |
| 30    | 72        | 78        | 98       |
| 37    | 80        | 86        | 108      |
| 45    | 88        | 94        | 118      |
| 52    | 94        | 102       | 128      |
| 60    | 102       | 110       | 138      |
| 67    | 110       | 118       | 148      |
| 75    | 116       | 126       | 158      |
| 82    | 124       | 134       | 168      |
| 90    | 132       | 142       | 178      |
| 97    | 138       | 150       | 188      |
| 105   | 146       | 158       | 198      |
| 112   | 152       | 166       | 208      |
| 120   | 160       | 174       | 218      |

## FPGA-to-FPGA Trace Delay Table

Trace delay is predictable, because it is a function of HT3 cable length. Trace delay here refers to more than just cable delay; it includes the FPGA pin-to-connector, cable delay, and connector-to-FPGA-pin delay.

The following table shows approximate delay values for different CON\_CABLE lengths. For the exact values for your hardware setup, consult the appropriate hardware documentation, which lists cable delays.

| CON_CABLE Length (cm)  | 25 | 50 | 100 | 150 | 200 | 300 | 400 |
|------------------------|----|----|-----|-----|-----|-----|-----|
| Total Trace Delay (ns) | 4  | 5  | 7   | 10  | 12  | 16  | 21  |

Add the total trace delay to the HSTDMDelayTables, on page 387.

## Mixed Configurations

For a hardware configuration with a mixture of systems, the tool uses a HSTDMD scheme that is supported by both systems in the configuration. In a mixed HAPS-80 and HAPS-70 configuration for example, the tool uses differential HSTDMD because it is supported by both systems. It only uses single-ended HSTDMD for hardware configurations composed of multiple HAPS-80 systems.

## Analyzing HSTDMD Results

Use this procedure to check the results after HSTDMD.

1. After routing, go to the `vivadoImplementation/HSTDMD_Qor_rpt` directory.

The tool generates reports from the post-routing DCP database, and puts them in the `HSTDMD_Qor_rpt` directory. If the tool cannot open the DCP files, it does not generate the reports.

The directory contains two kinds of reports:

- Clock-oriented reports, which report problems between two clocks
- Channel-oriented reports, which indicate HSTDMD channels with problems.

2. Check the following clock-oriented reports:

`clock_summary.rpt`

Generated if the design contains HSTDMD logic.

|                                                                                      |                                                                                                                                                       |
|--------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>clock_summary_from_to_HSTDm_path_issues.rpt</code>                             | Generated if there are timing violations between user and HSTDm logic. Paths between UMR clock and HSTDm clock are considered as from/to HSTDm paths. |
| <code>clock_summary_intra_HSTDm_path_issues.rpt</code>                               | Generated if there are timing violations within HSTDm logic. It also includes multi-hop paths (from HSTDm receiver to sender).                        |
| <code>timing_from_to_hstdm</code> and<br><code>timing_intra_hstdm</code> directories | Detailed clock-to-clock reports.                                                                                                                      |

Report names in the `timing_from_to_hstdm` and `timing_intra_hstdm` directories follow this scheme: `timing_fromClk_to_toClk<Suffix>.rpt`. The report does not have a *suffix* if there are no problems. The table lists the suffixes:

| Suffix                     | File Description                                                                                     |
|----------------------------|------------------------------------------------------------------------------------------------------|
| <code>_VIOLATED_max</code> | Hold timing violations between clocks                                                                |
| <code>_VIOLATED_min</code> | Setup timing violations between clocks                                                               |
| <code>_ASYNC</code>        | Untimed paths and unapplied constraints because the paths are between clocks in asynchronous groups. |

Check the following reports in the `timing_intra_hstdm` directory.

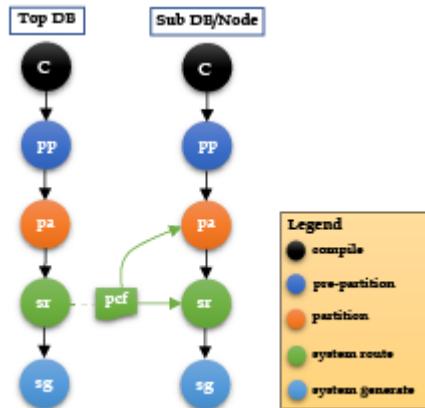
| File                                                       | Description                             |
|------------------------------------------------------------|-----------------------------------------|
| <code>timing_cpm_rcv_to_cpm_snd&lt;suffix&gt;.rpt</code>   | Multi-hop path report                   |
| <code>timing_hstdm_mux_from_rcv&lt;suffix&gt;.rpt</code>   | Timing report for HSTDm receiver MUX    |
| <code>timing_hstdm_mux_to_snd&lt;&lt;suffix&gt;.rpt</code> | Timing report for HSTDm transmitter MUX |

3. Check the following reports in the `HSTDm_constraints` directory for problems with the HSTDm channels:

| File                                 | Description                                                                                                                                                                                        |
|--------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| HSTDMD_constraints_async_clk_grp.xdc | Reports asynchronous clock group problems.                                                                                                                                                         |
| HSTDMD_constraints_missing.xdc       | Reports missing constraints if the Report Missing Constraints option was enabled. problem. This file reports the name of the HSTDMD receiver/sender module and the pin index of the problem paths. |
| HSTDMD_constraints_too_tight.xdc     | Report missing constraints that cause timing violations. This file reports the name of the HSTDMD receiver/sender module and the pin index of the problem paths.                                   |

## Using HSTDMD Top I/O Flow

The HSTDMD top I/O flow allows you to work on different sub-designs in parallel with the top-level design and then place them together. To set up a top I/O flow, identify the modules (sub-systems) and mark them as top I/O, then run the regular flow on the two databases/nodes.



Follow these steps to run the top I/O flow on a design.

1. Identify the modules to be selected as candidates for a top I/O flow from the top-level design.

2. Mark the modules as black boxes in the compile stage using this compiler directive

```
define_directive {v:subsystemName} {syn_subsystem} {1}
```

3. Run compile on the top-level design.
4. Run pre-partition on the top-level design. Use the same TSS file for implementing both the main and the sub-system databases.
  - In the partition stage include a PCF file defining the module identified as a top I/O.

```
subsystem_top_io -cell subsystemName -bin fpgaBins
assign_cell mainInstance fpgaBins
```

```
#set -ignore to 1 to ignore global clocks
subsystem_top_io -pin subsystemClk -ignore 1
```

- The FPGA bins selected for implementing the sub-system are dedicated to the sub-system. Do not assign any other logic to those FPGAs.
5. In the system route step use the same PCF file as for partitioning.
  - Always use the slack mode optimization priority so that the tool exports timing information.
  - Export two PCF files for the `system_route` node which will be used while implementing the sub-system database/node.

```
#contains clock and timing information
subsystem_top_io_ subsystemViewName_clk.pcf
```

```
#containing HSTDMD trace/ratio information
subsystem_top_io_ subsystemViewName_loc.pcf
```

6. Run system generate.

Bit files will be generated for the main design FPGAs.

## Subsystem Database/Node

1. Set the sub-system as top-level and run compile.
2. Run pre-partition using the same TSS file used for implementing the main design above.

3. In the partition stage include the PCF file exported from the system route stage of the main design.

This file defines the HSTDIM trace, ratio, and clock information of the main design.

Lock all other FPGA bins except the FPGAs used while running the sub-system's partition flow. Use the following pcf command to lock all other FPGAs automatically except those specified in the command.

```
subsystem_top_io -use_bins {subsystemFpgas}
```

4. In the system route step use the same PCF file exported from the system-route stage of the main database.

5. Run system generate.

Bit files are generated for the sub-system FPGAs

## PCF Commands for Setting Up Top I/O (Team Based Design) Flow

To define subsystems:

```
subsystem_top_io -cell {subsystemInstanceName} -bin {fpgaBins}
```

To ignore global clock pins:

```
subsystem_top_io -pin {subsystemClk} -ignore 1
```

To lock all unused FPGAs automatically:

```
subsystem_top_io -use_bins {subsystemFpgas}
```

## Exporting the PCF File from the Tool

The PCF file exported from the tool contains routing and timing information.

For subsystem\_top\_io\_subsystemViewName\_clk.pcf:

```
subsystem_top_io $types(instanceName) [objName {instanceName} {netName} -
bin {fpgaBin}
-qualified <tdm_qual 1/0> -clock_event {clkName:edge:delay:hop}
```

For subsystem\_top\_io\_subsystemViewName\_loc.pcf:

```
subsystem_top_io $types(instanceName) [objName {instanceName} {netName} -
bin {fpgaBin}
-qualified <tdm_qual 1/0> -trace {traceName} -tdm_slot {slotName} -tdm_module
{tdmRatio}
```

## Limitations

Currently, there are some limitations to the HSTDMD Top I/O flow.

- Gated clock and reset passing from main design to sub-system.
- Identification of cut clocks.
- Reset synchronization across subsystem FPGAs (both from top-level port and reset-generator net).
- Bi-directional bus passing across sub-system and the main design.
- A complete timing picture is not available since the system is not presently timing aware.
- Clock gate replication in the sub-system FPGA.
- SLR partitioning.
- Sub-system FPGA feedthroughs.
- Reports requiring information from the sub-system will not be correct.
- Unified Compile flow with Verilog wrapper over VHDL RTL.

## Using Asynchronous Pin Multiplexing (ACPM)

The ACPM (asynchronous pin multiplexing) model is MUX-based, with a fast clock for global distribution, and a control signal for data synchronization. It uses single-ended signaling and is not as fast as the HSTDMD model ([Using HSTDMD and HSTDMD ERD, on page 376](#)). With ACPM, the tool automatically assigns the sync signal and creates timing constraints, trace assignments and Confpro hardware settings.

The following steps describe controls for ACPM.

1. Set general TDM settings, as described in [Setting General Controls for TDM, on page 373](#).
2. Define the fast clock for ACPM in the pcf file.

ACPM uses a single fast clock.

- Use `tdm_control -acpm_fast_clock_trace` to define the fast clock.
- Configure the specified trace as a global clock in the tss file. You cannot specify a hardware global clock sourced from an FPGA.
- Set the frequency for the fast clock with `tdm_control -acpm_fast_clock_freq`. This frequency is shared between different ACPM ratios.
- 

## Using Multi-Gigabyte Transceivers for TDM

To significantly improve TDM speeds on HAPS-100 modules, use the multi-gigabyte transceivers (MGBs). You can run MGTDM (multi-gigabyte TDM) automatically or manually. Additionally, there are two modes to choose from: default mode at 12.5 Gbps, or performance mode at 20 Gbps.

1. Define the MGB2 connections for MGB2 quads in the TSS file.

A channel is 1 GTY/GTX transceiver (1 Tx and 1 Rx pair). A quad is a group of two GTY/GTX transceivers.

- To automatically use fixed traces, do not specify anything. There are 5 fixed quads per HAPS-100 module for fixed transceiver interconnect, but only quad 226 is used for MGTDM. Quad 226 is a full quad with four channels; quads 220, 225, 230 and 235 are half quads.
- In addition to the fixed traces, you can also specify connectors from the MGB2 quad connectors. HAPS-100\_4F modules have 6 quads connected to MGB2, so there are 24 MGB2 connectors available for MGTDM in addition to the fixed interconnect. All MGB2 quads are full quads, with 4 channels per MGB2. The quads that are connected to MGB2 are shown below.

### FPGA MGB Quad Connectors

|   |                                          |
|---|------------------------------------------|
| A | AM221, AM222, AM231, AM232, AM236, AM237 |
| B | BM221, BM222, BM231, BM232, BM236, BM237 |
| C | CM221, CM222, CM231, CM232, CM236, CM237 |
| D | DM221, DM222, DM231, DM232, DM236, DM237 |

Use the connector names from the previous table to manually specify MGTDM cables in the TSS file. Do not specify fixed quad 226, which the tool uses automatically. For example:

```
board_system_create -interconnect -manual CON_CABLE_100_MGB2
 -name mgb1 -connector {FB1.AM221 FB1.BM221}
```

Make sure to use the correct cable for the chosen speed grade and planned MGTDM mode. See this table:

**Speed Grade**

| FPGA 1 | FPGA2 | Cable Length | Performance Mode | Default Mode |
|--------|-------|--------------|------------------|--------------|
| -1     | -2    | 100 cm       | Not Supported    | Supported    |
| -1     | -2    | 200 cm       | Not Supported    | Supported    |
| -1     | -1    | 100 cm       | Not Supported    | Supported    |
| -1     | -1    | 200 cm       | Not Supported    | Supported    |
| -2     | -2    | 100 cm       | Supported        | Supported    |
| -2     | -2    | 200 cm       | Not Supported    | Supported    |

- For MGTDM default mode (12.5 Gbps), specify these constraints in the PCF file:

- Enable MGTDM with this command:

```
tdm_control -enable_mgtdm 1
```

To disable MGTDM insertion, set the constraint to 0:

```
tdm_control -enable_mgtdm 0
```

- Also specify the inter-FPGA nets for MGTDM with PCF constraints:

```
net_attribute {list_of_nets} -function MGT -tdm_group
 {MGTDM_module_with_ratio}
```

The MGTDM module is called HSTDMD\_MGT. These are examples of the PCF constraint:

```
net_attribute {u4.dat1[39:0] u4.dat2[39:0] u4.dat3[39:0]} -function MGT
 -tdm_group HSTDMD_MGT_256
net_attribute {u4.dat1[39:0] u4.dat2[39:0] u4.dat3[39:0]} -function MGT
 -tdm_group HSTDMD_MGT_128
```

The MGTDM delays for various TDM ratios in default and performance modes are shown below:

| <b>Ratio (Fixed Traces)</b> | <b>MGTDM Delay: Default</b> | <b>MGTDM Delay: Performance</b> |
|-----------------------------|-----------------------------|---------------------------------|
| 64                          | 95                          | 63                              |
| 128                         | 103                         | 69                              |
| 256                         | 113                         | 75                              |
| 512                         | 133                         | 87                              |
| 1024                        | 173                         | 113                             |

3. For MGTDM performance mode (20 Gbps), set this PCF constraint instead of the ones described in the previous steps:

```
tdm_control -low_latency_mgtdm 1
```

4. To run MGTDM in automatic mode, do the following:

- Specify this constraint in the PCF file:

```
tdm_control -enable_mgtdm auto
```

The tool runs MGTDM automatically in default mode, using the fixed trace (quad 226) for MGTDM.

- To run the tool automatically in performance mode, specify these PCF commands:

```
tdm_control -enable_mgtdm auto
tdm_control -low_latency_mgtdm 1
```

5. Run system route with the following optimization options:

```
run system_route -aptn_srp_build 1 -optimization_priority slack
-mapped_timing_models 1
run system_route -optimization_priority multi_hop_path -estimate_timing 1
```

6. Check results.

- If the design was run with distributed synthesis, you can check the constraints\_application.srr log file to see which quads are used. If distributed synthesis was not used, this information is available in the pre-map log file, pre\_map.srr.

| Multi-gigabit transceiver TDM report for FB1_uA |         |           |            |                |       |            |
|-------------------------------------------------|---------|-----------|------------|----------------|-------|------------|
| =====                                           |         |           |            |                |       |            |
| MGTDM Info:                                     | Quad_id | Direction | Channel_Id | Bit rate(Mbps) | Ratio | Trace Type |
|                                                 | 221     | Receive   | 3          | 12500          | 256   | MGB2       |
|                                                 | 221     | Transmit  | 0          | 12500          | 256   | MGB2       |
|                                                 | 221     | Transmit  | 2          | 12500          | 256   | MGB2       |
|                                                 | 221     | Receive   | 1          | 12500          | 256   | MGB2       |
|                                                 | 221     | Receive   | 2          | 12500          | 256   | MGB2       |
|                                                 | 221     | Transmit  | 1          | 12500          | 256   | MGB2       |
|                                                 | 221     | Transmit  | 3          | 12500          | 256   | MGB2       |
|                                                 | 221     | Receive   | 0          | 12500          | 256   | MGB2       |
|                                                 | 222     | Transmit  | 3          | 12500          | 256   | MGB2       |
|                                                 | 222     | Receive   | 1          | 12500          | 256   | MGB2       |
|                                                 | 222     | Transmit  | 1          | 12500          | 256   | MGB2       |
|                                                 | 222     | Receive   | 3          | 12500          | 256   | MGB2       |
|                                                 | 222     | Transmit  | 2          | 12500          | 256   | MGB2       |
|                                                 | 222     | Transmit  | 0          | 12500          | 256   | MGB2       |
|                                                 | 222     | Receive   | 0          | 12500          | 256   | MGB2       |
|                                                 | 222     | Receive   | 2          | 12500          | 256   | MGB2       |
|                                                 | 226     | Transmit  | 0          | 12500          | 256   | FIXED      |
|                                                 | 226     | Receive   | 3          | 12500          | 256   | FIXED      |
|                                                 | 226     | Receive   | 2          | 12500          | 256   | FIXED      |
|                                                 | 226     | Transmit  | 2          | 12500          | 256   | FIXED      |

- Check the Global Route Ratio Report (AP366) for the MGTDM ratio, trace usage, and bit rate.

@S3.1 AP267 |Post-Optimization and Cable Assignment Global Route Report

@S3.1.1 AP366 |Global Route Ratio Report

| Connection            | Trace_Usage | Net_Usage | TDM | DIRECT | CLOCK    | Ratios | Bitrate | Cablelength |
|-----------------------|-------------|-----------|-----|--------|----------|--------|---------|-------------|
|                       |             |           |     |        |          |        | in Mbps | in cm       |
| FB1.uA<->FB1.uB       | 25/26       | 0         | 25  | 0      |          |        |         |             |
| FB1.uA<->FB1.uB (MGT) | 4/8         | 1920      | 0   | 0      | 512      |        | 12500   |             |
| FB1.uC<->FB1.uD       | 0/81        | 0         | 0   | 0      | Not used |        |         |             |

- Check the post-optimization routing summary (AP635) for details about the TDM module and post-TDM clock range.

| Connection      | Trace Usage | No. of Net failed timing | TDM Module              | Available Slack (Post TDM) | Range   | Mean Slack |
|-----------------|-------------|--------------------------|-------------------------|----------------------------|---------|------------|
| FB1.uA<->FB1.uB | 25/26       | 0/25                     | NONE (4 ns)             | 1190 to 1195 ns            | 1190 ns |            |
| FB1.uA<->FB1.uB | 4/8         | 0/1920                   | HSTDMD_MGT_512 (132 ns) | 1066 to 1066 ns            | 1066 ns |            |

## 7. Run TDM training.

- Create a JSON file for TDM training. Use the format described in [Creating an HMF File, on page 379](#), modifying it for your

requirements. Run training using the proto\_rt command and specifying the json file:

```
proto_rt::run_ipinfra -train all -hmf training.json
```

- Generate a report and analyze the results using these commands. For example:

```
proto_rt::run_ipinfra -hmf training.json -report all
```

```
proto_rt::run_ipinfra -report_verbose all -hmf training.json -file report.txt
```

See [Using proto\\_rt for HSTDMD Training, on page 377](#) for details about using this command.

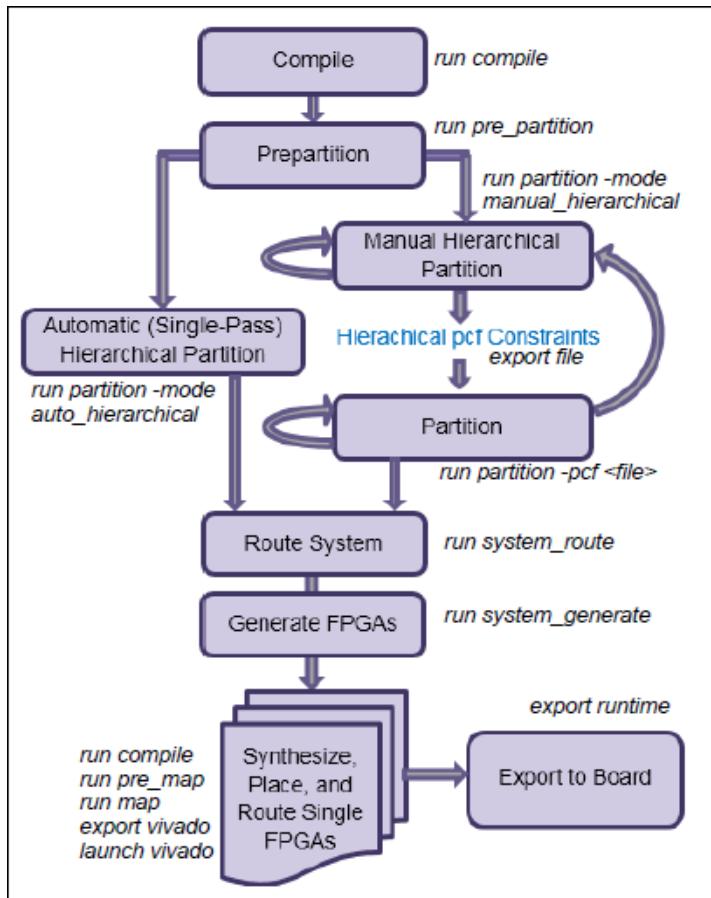
## MGTDM Limitations

- MGTDM is not supported for mixed-system (HAPS-100 and HAPS-80) designs.
- Half quads (220, 225, 230 and 235) are not used for MGTDM because these channels are shared with debug channels.
- System-level timing analysis is not supported with MGTDM.
- Note that the 200 cm MGB2 cable is not supported for performance mode.

# Using Hierarchical Partitioning

For complex prototypes with eight FPGAs or more, use hierarchical partitioning to break down the problem into smaller pieces. Hierarchical partitioning is based on the concept of super-bins. A super-bin consists of the FPGAs on a particular *board* on the prototype. The idea is to first partition the multi-FPGA prototype into board-level super-bins, and then use the constraints generated by this first step to complete partitioning down to the FPGA level.

You can run hierarchical partitioning automatically or manually. In automatic mode, the tool runs single-pass recursive partitioning instead of the two-pass methodology required for manual mode.



See the following topics for details:

- [Running Hierarchical Partitioning in Automatic Single-Pass Mode](#), on page 408
- [Running Top-Down Hierarchical Partitioning in Manual Mode](#), on page 409

## Running Hierarchical Partitioning in Automatic Single-Pass Mode

The steps to run top-down hierarchical partitioning automatically, in a single-pass recursive mode are described below. For information about running hierarchical partitioning in two-pass manual mode instead of automatic mode, see [Running Top-Down Hierarchical Partitioning in Manual Mode, on page 409](#).

1. Define the configuration in the TSS file and complete the compile and pre-partition stages.
2. Define FPGA superbins.

By default the partitioner creates *superbins* from each “board” when it runs in hierarchical mode. You can use either a basic or detailed TSS file, depending on where you are in the design cycle. The following TSS definition automatically creates superbins for FB1 and FB2:

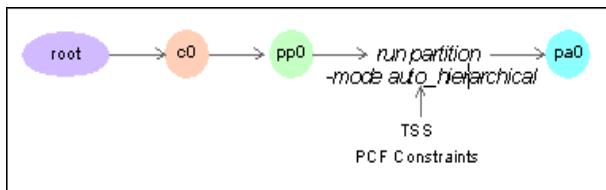
```
board_system_create -add HAPS-70_S48 -name FB1 # Creates FPGA bins
 {FB1.u[A-D]}
board_system_create -add HAPS-70_S48 -name FB2 # Creates FPGA bins
 {FB2.u[A-D]}
```

You can change the defaults and define your own superbins using pcf constraints:

```
hierarchical_super_bin -name g1 {fb1.uA fb1.uB fb1.uC}
hierarchical_super_bin -name g2 {fb1.uD fb2.uA fb2.uB}
```

3. Run partition, specifying automatic hierarchical partitioning with the **-mode** argument.

```
run partition -mode auto_hierarchical -tss tss1.tcl -pcf clocks.pcf -pcf
ports.pcf
```



The tool automatically runs recursive hierarchical partitioning and generates a constraint file with port and cell assignments.

4. To reuse these constraints, export the file:

```
export file assignments.pcf
```

5. Continue with the rest of the partition flow, by routing system traces and generating FPGA partitions.

For additional information about the commands and tasks mentioned in the preceding procedure, refer to the following topics:

- [Partitioning the Logic](#), on page 328
- [Target System Specification Commands](#), on page 243 in the *Command Reference Manual*
- [run partition](#), on page 144 in the *Command Reference Manual*

## Running Top-Down Hierarchical Partitioning in Manual Mode

The steps to run top-down hierarchical partitioning in a two-pass manual mode are described below. For information about running hierarchical partitioning in single-pass recursive mode instead of manual mode, see [Running Hierarchical Partitioning in Automatic Single-Pass Mode](#), on page 408.

1. Define the configuration in the TSS file and complete the compile and pre-partition stages.
2. Define FPGA superbins.

By default the partitioner creates *superbins* from each “board” when it runs in hierarchical mode. You can use either a basic or detailed TSS file, depending on where you are in the design cycle. The following TSS definition automatically creates superbins for FB1 and FB2:

```
board_system_create -add HAPS-70_S48 -name FB1 # Creates FPGA bins
 {FB1.u[A-D]}
board_system_create -add HAPS-70_S48 -name FB2 # Creates FPGA bins
 {FB2.u[A-D]}
```

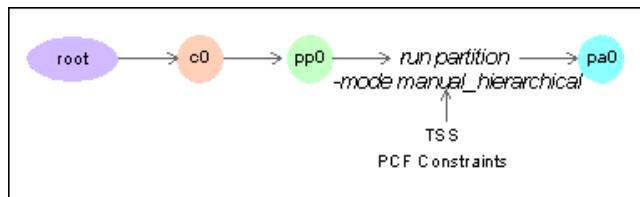
You can change the defaults and define your own superbins using pcf constraints:

```
hierarchical_super_bin -name g1 {fb1.uA fb1.uB fb1.uC}
hierarchical_super_bin -name g2 {fb1.uD fb2.uA fb2.uB}
```

3. Run a first pass of hierarchical partitioning.

Start with a pre-partitioned database state, and run the partitioner in manual hierarchical mode (-mode) and the appropriate TSS and pcf files. For example:

```
run partition -mode manual_hierarchical -tss my_tss.tcl -pcf GCLKS.pcf
-pcf abstract.pcf
```



After this pass of hierarchical partitioning, the tool generates a constraint file that constrains all the cells in the design to the bins that make up the superbins. The output for a cell is constrained by its assignment to a superbin and any user constraints you might have set. Replication constraints are directly copied to the output constraints. The superbin assignments are in the `cell_constraints.pcf` file.

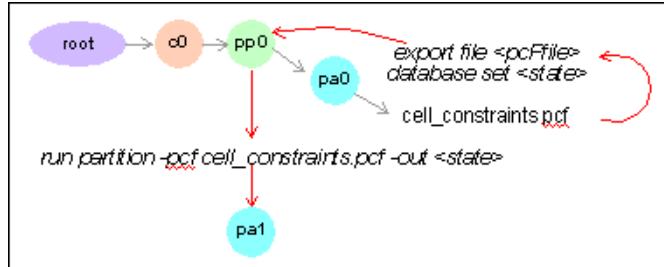
4. To reuse hierarchical files generated from this database state, first export the constraints:

```
export file cell_constraints.pcf
```

- ## 5. Rerun partitioning with the hierarchical constraints.

The following example shows the commands used to rerun partitioning, with the red arrows indicating the second run with reused constraints:

```
database set pp0
run partition -pcf cell_constraints.pcf -out pa1
```



- 
6. Continue with the rest of the partition flow by routing system traces and generating FPGA partitions.

For additional information about the commands and tasks mentioned in the preceding procedure, refer to the following topics:

- [Partitioning the Logic](#), on page 328
- [Target System Specification Commands](#), on page 243 in the *Command Reference Manual*
- [run partition](#), on page 144 in the *Command Reference Manual*

# Running System Route for the Top Level

After you have partitioned the design, the next step is system routing at the inter-FPGA level. During this phase, the tool assigns inter-FPGA nets to HAPS board traces and performs TDM as needed. This is the stage where the partitions are actually created. After you route the system level for the first time, the tool generates a database state called sr0.



You can skip this step and go directly from partitioning to generating the individual FPGA partitions, but you might have more pins than are allowed. You can skip this step if you are exploring different solutions.

1. Start with a partitioned database state that meets your goals.

See [Partitioning the Logic, on page 328](#) for details.

2. Gather other input for system routing.

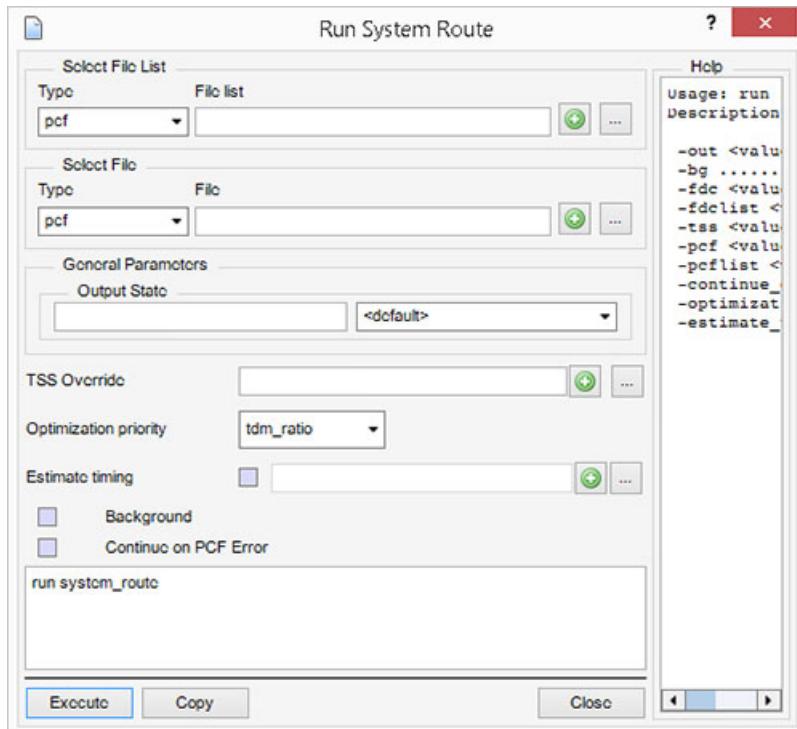
The files mentioned below are specified from the command line in the next step.

- Use the target system information (tss file) from the report generated after partitioning to see the pin and trace name assignments.
- Specify routing constraints in a pcf file. See [Specifying PCF Constraints for Partitioning, on page 245](#). Add constraints as needed, but do not make drastic changes or the results might not be routable. For example, add keepbufs for feedthroughs, add net attributes to control post-partition routing, separate ports and nets for nets that will be routed on two different channels, and define end pins for differential clocks.

System route requires that a pcf file with clock and reset information be specified.

- Optionally, specify an FDC file with constraints for compiling and mapping the partitions. Generates a syntax\_constraint\_check.rpt file with the report constraint\_check -syntax\_only command, and make sure that all constraint syntax has been specified correctly.

- Set other options as needed in a Tcl options file, and source the Tcl file.
3. Use the run system\_route command to create system-level routing.
- To use the GUI to run the command, start with a partitioned database, click System Route, and specify the command parameters. For descriptions of the dialog box options, see [Run System Route Dialog Box, on page 111](#) in the Reference Manual.



- If you have pcf constraints, specify them with the -pcf argument or in the appropriate field in the GUI.
- Optionally, specify an fdc constraints file. Some constraints take effect at this stage.
- To specify timing-aware optimizations for TDM and multi-hop paths, use the -estimate\_timing and -optimization\_priority multi\_hop\_path arguments with the run system\_route command. See [Setting General Controls for TDM, on page 373](#) for a tabular description of the effect of these options. Use the net\_attribute -tdm\_qualified command in the pcf file to

override these general option settings and qualify nets for TDM on a per-net basis.

See [run system\\_route, on page 160](#) in the *Command Reference* for details about the command and its arguments.

The tool routes the inter-FPGA traces for the whole system. This includes TDM with ratio assignment and grouping, and detailed routing assignments. If you specified multi-hop optimizations, it runs timing-aware optimizations to reduce the number of hops between FPGAs.

#### 4. Analyze results.

- Specify the report target\_system -tss *tssFile.tcl* command to generate the report file. Run view target\_system.rpt to view the report. For more information, see [Checking the Target Specification \(TSS\), on page 358](#).
- Check the slack value in system\_route.rpt for each net.
- Check timing.rpt for updated values. The report contains FPGA names and timing.
- Check the updated TDM report: tdm\_qualified\_timing.rpt.

For more information about partitioning reports, see [Analyzing Partition Result Files, on page 358](#) and [Optimizing Multi-Hop Paths, on page 344](#).

#### 5. To freeze partition results, export the router\_constraints.pcf file and use it on the next run, as described in [Locking Down Partitions, on page 352](#).

## Congestion Reduction Using Pin Table Assignment

### Beta

After partitioning, use option set enable\_conn\_table 1 to reduce congestion. To use this option, set the license\_feature FpgalInternalBeta.

In this mode, the ProtoCompiler tool creates a FPGA pin connectivity table based on the design. This includes connectivity crossing flip-flops and D/Q pins. The tool calculates an estimate of the SLR crossings induced by the router trace assignment. The tool uses the cable optimizer algorithm after routing, to reduce estimated SLR crossings. Use the run system\_route command to optimize the SLR crossing:

```
run system_route -mapped_timing_models 1 -optimization_priority slack
run system_route -estimate_timing 1 -optimization_priority multi_hop_path
```

Check the system\_route.log for the SLR crossing enhanced report:

"@S4.7 AP612 |Cable Assignment Optimization " section is system\_route.log file is enhanced SLR crossing related info.

# Generating FPGAs

Generate the individual FPGA partitions when you are happy with the partitioning results. The system generate stage is the terminal step in the basic partitioning flow. At this stage, the tool automatically creates a timing budget and generates partitions based on the hardware specification in the tss file and the pcf and fdc constraints.



You can also return to the system generate stage after synthesizing individual FPGAs or to back-annotate place and route results for system-level timing analysis.

See the following for details:

- [Using system\\_generate to Generate FPGAs, on page 416](#)
- [Analyzing Results after System Generate, on page 418](#)
- [Working with Inferred Clocks Driving Inter-FPGA Paths, on page 424](#)

## Using system\_generate to Generate FPGAs

The following procedure describes how to generate individual FPGAs with run system\_generate.

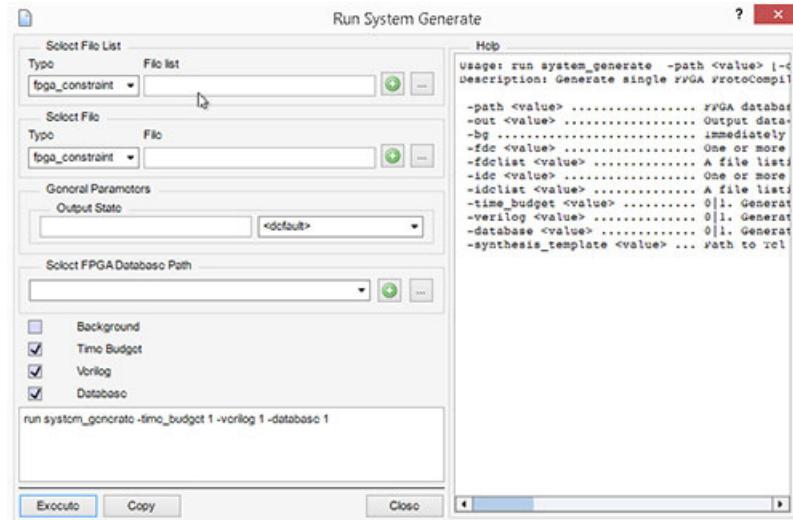
1. Start with a database state that has been partitioned (run partition) or where you have run system routing (run system\_route).
2. Specify other input.
  - Specify additional partitioning constraints as needed in a pcf file. See [Specifying PCF Constraints for Partitioning, on page 245](#). Specify the pcf file on the command line in the next step.
  - Set other options as needed in a Tcl options file. Source the Tcl file.
3. Use the run system\_generate command to partition the design between multiple FPGAs.

- Specify fdc constraints and debug point constraints (idc). You must specify the FDC file so that the generated partitions are constrained properly. Generate a `syntax_constraint_check.rpt` file with the `report constraint_check -syntax_only` command, and check that all FDC constraint syntax is correct.
- Set the `-time_budget` argument to 0 if you do not want timing budgets to be used.

See [\*run system\\_generate\*, on page 157](#) in the *Command Reference* for details about the command and its arguments.

The tool creates a directory for each FPGA. If you are using Multi-FPGA DTD, the tool generates an additional device called dtd2, which you must treat like any other FPGA. Each FPGA has its own single-chip database, HDL, and constraints. The directory also contains a file with paths to all generated and original HDL files, and a script to run the partitioned single-FPGA design through place-and route.

To use the GUI to run the command, start with a database state where system level routing is complete, click System Generate, and specify the command parameters. For descriptions of the dialog box options, see [\*Run System Generate Dialog Box\*, on page 113](#) in the *Reference Manual*.



Once you have completed the first round of system routing, you can implement individual FPGAs, as described in [Implementing Individual FPGAs, on page 426](#). You can also simulate the post-partitioned design using simulation-ready RTL logic models. See [Running Simulation, on page 557](#) for details.

You can also return to the system route state after synthesizing individual FPGAs for analysis and backannotation.

## Analyzing Results after System Generate

1. Use the report syntax\_constraint\_check command to check the HDL for the generated partitions.

View the syntax\_constraint\_check.rpt file for details.

2. Use the report timing command to generate timing results.
  - Use report timing -generate to generate the report. Timing analysis at this stage always reflects system-level timing analysis (SLTA), and only reports inter-FPGA paths. HSTDMP IPs are modeled as through points. The contents of the report vary, depending on the status of the system generate state:

| System Generate State         | Timing Report Contents               |
|-------------------------------|--------------------------------------|
| With time budgets             | Contains time budget estimates       |
| After mapping FPGAs           | Post-synthesis (mapped FPGAs) timing |
| After backannotating FPGA P&R | Post-P&R timing                      |

- Specify the kind of SLTA with the -mode argument. You can check what modes are available with report timing -list. Refer to [Running System-Level Timing Analysis \(SLTA\), on page 547](#) for a detailed procedure on running system-level timing analysis.
- Customize the timing report by using specific arguments with the report timing -generate command, like -from, -rise\_to, -slack\_margin, and so on. Refer to [report timing, on page 127](#) in the *Command Reference Manual* for the complete list of arguments.

You can also use get\_\* or find query commands to narrow down the report further. For example:

```
report timing -generate -from [get_cells {fb1.uB.wb_add[*]}]
```

When you use -from, -to, or -through filters, remember to use mapped names, not RTL names. RTL instance names can change during mapping; for example, an RTL RAM could have changed from {i:dest\_u4.ram1[7:0]} to {i:dest\_u4.ram1\_0\_0} after mapping.

3. Check timing budget results.

The set\_datapathonly\_delay constraint is used to model direct paths between partitions, instead of input and output delays. These constraints are set on I/O ports, for every clock event going through that port instead of just the most critical event. See [Time Budgeting, on page 146](#) in the *Reference Manual* and [set\\_datapathonly\\_delay, on page 317](#) in the *Command Reference* for details.

- Look for warnings about negative slack and negative timing constraints in the log and budget fdc constraint files for the FPGAs. A time budget of 0 indicates that the design is over-constrained and will not meet timing after the partitions are synthesized. This could show up as a requested period of 0 or a DPO of 0 down the line, after mapping.

Prevent this by checking the time budget log for slack details and for warnings. Some typical causes include clocks with misaligned edges between start and end points, missing timing exceptions between user clocks, large arrival times at the start point, or large output delays on the receive side.

There are some techniques to address time budget issues found at this stage. One method is to reduce the requested clock frequency. You can also work on the critical paths. Check the nets considered for TDM, and the TDM ratio used. If possible, remove the nets from TDM, and route them directly. If you need the performance, try adding more cables, or get the target frequency in the ballpark, and then explore advanced partitioning options.

- Check the UMR logic and set exception constraints (false path, multicycle path or max delay) as needed on UMR paths. You cannot change the UMR clocks, which the tool automatically inserts. The UMR clocks are listed in the Clock Summary section of the log file.
4. After compile and map, use the `check_hstdm_timing` utility to check for missing time budget constraints.

For information about using this utility, see [Generating and Viewing Timing Reports, on page 528](#).

5. Analyze results.

- Check for warning and error messages.
- Use the `view schematic time_budget` command to view the results in a schematic. Use `view schematic -list` to see a list of views available.

See [Analyzing Results, on page 512](#) for information about using analysis tools.

## Interactive Partitioning Flow

### Beta

The interactive partitioner is a Tcl shell in the HAPS automatic partitioner (APTN). You can refine the partitioning process by moving cells into multiple FPGAs and get reduce multihop paths.

### Phases in Interactive Partitioning

There are phases in interactive partitioning. Interactive partitioner can be used to fine tune partition and get impact of specific changes rather than running entire flow which will need more runtime. Obtain the `FpgaInternalBeta` license to enable this feature.

To invoke the interactive partitioner enable the option `-interactive 1`, and provide constraints into pcf files using the run partition command.

- The run partition command with the `-interactive` option activates the partitioner in interactive mode.
- The ipcf file contains information to run the APTN in interactive mode.

Example syntax:

```
run partition -tss abstract.tss -pcf setup.pcf -pcf abstract.pcf -
interactive 1 -ipcf ipcfFileName
```

- Constraint phase

In this phase, launch the interactive partitioner from the GUI or in batch mode, and provide the setup constraints using pcf files. The tool reads the inputs such as the netlist and the .pcf files, and the .ipcf file. The ipcf file contains information to run the APTN in interactive mode.

- Use the `run_flow` command to run the partitioner in interactive partitioning shell. It executes the partitioning algorithm and switches to the next phase, manual partitioning. The ipcf file starts with the `run_flow` command and end with the `exit` command.
- Manual partitioning phase

In this phase, make changes to the partition and analyze the results and the QoR impact. You can go through multiple iterations and refine the results until you are satisfied. The `-ipcf` option is used with the `run partition` command and it contains commands that are needed in the manual partitioning phase. The `-ipcf` option starts with `run_flow` command, which switches the partitioner into manual partitioning phase.

The exit command quits the interactive session.

Commands in this phase are different from commands in constraint phase and some of the commands that were available in constraint phase are not available in manual partition phase.

help is used to access the entire list of commands.

```
>help
assign_cell
assign_port
design_list_cells
design_list_nets
design_list_ports
design_list_properties
dissolve_cell
exit
fix_multihop_path
help
replicate_cell
report_cell_connections
report_cost
report_interconnection_table
report_solution
report_timing
run_clock_replication
run_flow
run_route
run_write_results
solution_commit
solution_discard
solution_restore
solution_save
tss_list_bins
tss_list_functions
tss_list_pins
tss_list_properties
tss_list_trace_groups
tss_list_traces
```

<command name> -help gives details of each command. For example:

```
run_flow -help
```

| <b>Flow commands</b> |                                                                                          |
|----------------------|------------------------------------------------------------------------------------------|
| <b>Command</b>       | <b>Description</b>                                                                       |
| run_flow             | Runs the partition algorithm and switches to manual partition phase.                     |
| run_route            | Runs the global router and prints summary (AP267)                                        |
| run_write_results    | Runs global route and srp build. Required to complete the flow.                          |
| exit                 | Quits interactive session. Returns success if run_write_results was called successfully. |

| <b>Solution commands</b>        |                                                                               |
|---------------------------------|-------------------------------------------------------------------------------|
| <b>Command</b>                  | <b>Description</b>                                                            |
| assign_cell {cell(s)} {bin}     | Assigns one or more cells to a FPGA bin in the temporary solution.            |
| assign_port {ports(s)} {bin}    | Assigns one or more ports to a Port bin in the temporary solution.            |
| replicate_cell {cell(s)} {bins} | Replicates one or more cells to multiple FPGA bins in the temporary solution. |
| solution_commit                 | Accepts all pending moves and applies them to the final solution.             |
| solution_discard                | Discards all the pending moves.                                               |
| solution_save {file}            | Saves the temp.solution into the file.                                        |
| solution_restore {file}         | Restores temp.solution from the file.                                         |

| <b>Report commands</b>       |                                                                                                 |
|------------------------------|-------------------------------------------------------------------------------------------------|
| <b>Command</b>               | <b>Description</b>                                                                              |
| report_cell_connections      | Reports cell connections similar to partition.rpt.                                              |
| report_solution              | Reports bin usage summary (AP140).                                                              |
| report_cost                  | Reports solution cost (depending on the optimization priority).                                 |
| report_interconnection_table | Prints information about interconnections and their tdm qualification, also saved to .csv file. |

| report timing                                                                                                                                                                | Reports timing. -type <hops> reports hops.<br>-type <value> ..... hops   slack (default hops).<br>-worst_paths <value> ... report worst paths.<br>-level <value> ..... report multihop paths of length n.<br>-net <value> ..... report worst paths for specified net.<br>-summary <value> ..... report summary.<br>-file <value> ..... write report to file. |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>List commands</b>                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                              |
| design_list_* and tss_list_* commands lists various properties as mentioned in the command. For details, check the command using -help. For example: design_list_cells -help |                                                                                                                                                                                                                                                                                                                                                              |
| <b>Fix multihop command</b>                                                                                                                                                  |                                                                                                                                                                                                                                                                                                                                                              |
| Command                                                                                                                                                                      | Description                                                                                                                                                                                                                                                                                                                                                  |
| fix_multihop_path -net name                                                                                                                                                  | The tool automatically attempts to fix worst multihop for the given net.                                                                                                                                                                                                                                                                                     |

## Working with Inferred Clocks Driving Inter-FPGA Paths

During time budgeting, there could be inferred clocks for paths between FPGAs. If there are many inferred clocks clocking inter-FPGA paths then these clocks and DPOs associated with them could cause long runtime issues in Vivado place and route.

There are two flows when working with inferred clocks clocking inter-FPGA paths. The default flow infers a clock for every unconstrained clock object in the design and the alternative flow merges clocks inferred for every clock object into one inferred clock. The following procedure shows how to use both flows.

### Default flow

1. After running pre-partition, check the clock information.
  - Specify clock definitions with `create_clock` for any inferred clocks.
2. Do the following for the default flow:
  - `-option set merge_inferred_clocks 0` (setting is default)
  - Partition the design as usual.

- Generate timing budgets at the System Generate stage (run `system_generate -time_budget 1`). With the default settings (option set `merge_inferred_clocks 0`), the tool infers multiple inferred clocks (as necessary) for unconstrained clocks clocking inter-FPGA paths. It creates constraints (`create_clock`) for each of them that are passed to the individual FPGAs.

```
create_clock -name {test|c1} -internal 6 -period 1000.00 -add [get_ports {c1}]
create_clock -name {test|c2} -internal 6 -period 1000.00 -add [get_ports {c2}]
create_clock -name {test|c3} -internal 6 -period 1000.00 -add [get_ports {c3}]
```

3. Check for DE107 errors. These errors indicate inter-FPGA paths that are clocked by inferred clocks. You can downgrade this error to a warning, so that you can continue. To downgrade the error, use the TCL command `message_override -warning DE107` and rerun System Generate. Or right-click on the error message and choose the option **Downgrade to Warning**. If you downgrade the error, the tool generates DPOs for the paths clocked by inferred clocks, in addition to the `create_clock` constraints.

## Merged Flow

With this option, the tool merges clock nets, ports, and pins into only one inferred clock, and creates a clock constraint. This constraint is passed to the individual FPGAs for use during synthesis and place and route.

To merge inferred clocks and generate only one inferred clock for the whole design, do the following:

1. Set option command

```
option set merge_inferred_clocks 1
```

2. Inferred clocks are forward annotated to Synthesis with the argument to its constraint `-internal 6`.

```
create_clock -name {test|c1} -internal 6 -period 1000.00 -add [get_ports {c1} {c2}
{c3} {c4} ...]
```

3. Check for DE107 errors, which can be downgraded as described in the [Default flow, on page 424](#). If you downgrade the error, the tool generates DPOs for the paths clocked by inferred clocks, in addition to the `create_clock` constraints.

# Implementing Individual FPGAs

Once you have partitioned your top-level prototype, you can synthesize the individual FPGAs. See the following for details:

- [Running Synthesis for Individual FPGAs](#), on page 426
- [Customizing Scripts to Synthesize, Place, and Route FPGAs](#), on page 430
- [Updating the Top Level with FPGA Implementation Results](#), on page 435

To reduce runtime, you can use distributed processing to work on the FPGAs in parallel. See [Running Distributed Synthesis](#), on page 506.

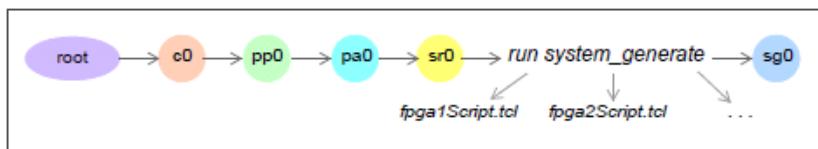
## Running Synthesis for Individual FPGAs

There are two ways to implement the individual FPGA partitions, an automatic, script-based method, and a manual one. Both methods are described below, but it is recommended that you use the first procedure.

### Synthesizing Individual FPGAs Using a Script

To use the recommended, script-based method to synthesize FPGAs, follow these steps:

1. Generate individual FPGA partitions with `run system_generate -path` ([Generating FPGAs](#), on page 416).



This command generates Tcl scripts for implementing the individual FPGAs. The `-path` option specifies a location for the Tcl scripts. For example, `run system_generate -out g0 -path synthesis_files` generates the following Tcl files for a design with two partitions, uA and uB:

```

synthesis_files/fb1_uA/fb1_uA_srs.tcl
synthesis_files/fb1_uB/fb1_uB_srs.tcl

```

The scripts include commands to compile, pre-map, and map the FPGAs. Each FPGA script also includes the `export vivado` command to generate the files needed for place and route.

You can customize the scripts as described in [Customizing Scripts to Synthesize, Place, and Route FPGAs, on page 430](#). However, do not change the speed grade or technology that are written to these Tcl files, because the partitions use this information. The speed grade for individual FPGAs is set with the `board_system_create` command in the TSS file, and the technology is determined by the `database create -technology` command.

2. To run the individual databases in parallel, specify the `launch protocompiler -script` command:
  - Use this basic syntax to process each script and implement the FPGAs in parallel. Refer to [launch protocompiler, on page 71](#) in the *Command Reference Manual* for the complete syntax description.

```
launch protocompiler -script fpga1.tcl -script fpga2.tcl ...
```
  - You can also set options for individual FPGA runs from the command line by preceding the `-script` argument with a `-tclcmd` argument. See [Customizing Scripts to Synthesize, Place, and Route FPGAs, on page 430](#) for details.
  - If you are using deep trace debug, make sure to implement the DTD device generated after the system generate step, just as you do with all the other FPGAs.
3. To run FPGA implementation scripts in parallel and minimize the overall runtime for a multi-FPGA design, do the following:
  - Specify the `launch protocompiler` command with the `-max_parallel_jobs` option to run individual FPGA implementations in parallel. You can run four jobs in parallel with each license. The actual number of jobs run will depend on the number of available licenses and the value specified.

```
launch protocompiler -max_parallel_jobs 8 -script fpga1.tcl -script fpga2.tcl ...
```

If you do not specify this argument, the command will use the default `max_parallel_jobs` value, or a value that was previously set. For more about specifying parallel jobs, see [Setting Number of Parallel Jobs, on page 612](#).

You can override the `max_parallel_jobs` setting used with the `launch` command by setting an option in an individual FPGA during synthesis, after the `launch` command has been issued.

- To allocate different values for parallel jobs for different FPGAs, specify `-max_parallel_jobs` before that FPGA script.

```
launch protocompiler -max_parallel_jobs 4 -script fpga1.tcl
 -max_parallel_jobs 8 -script fpga2.tcl ...
```

- To use CDPL as the basis for distributed processing, set it up as described in [Using CDPL for Distributed Processing, on page 500](#). If you do not use CDPL, the tool will use the available processors on the current machine to run the parallel processes.
- Specify the scripts to run in parallel when you specify the `launch` command, as shown above. The tool divides the jobs and uses distributed processing.
- You can also set options for the run from the command line. See [Customizing Scripts to Synthesize, Place, and Route FPGAs, on page 430](#) for details.
- If you are using deep trace debug, make sure to implement the DTD device generated after the system generate step, just as you do with all the other FPGAs.

4. Run place and route for individual FPGAs with the `launch vivado` command.

The script generates files for the Vivado run with the `export vivado` command.

- To use the default Vivado Tcl script for place and route, run Vivado with commands like the following example, where `myVivadoDir` is the directory generated by the script after running `export vivado`:

```
set vivado_tcl_script [export vivado -path myVivadoDir]
launch vivado $vivado_tcl_script -run_dir $fb_uA1_dir
```

- To use a custom place-and-route script, specify it instead of using the Tcl script generated by `export vivado`, as shown here:

```
launch vivado /home/my_custom_vivado_script.tcl -run_dir $fb_uA1_dir
```

This is an example of the command sequence:

```
launch protocompiler -max_parallel_jobs -script path/slp1.tcl -script path/slp2.tcl
 -script path/slp3.tcl
```

```
set vivado_tcl_script [export vivado -path myvivadodir]
launch vivado $vivado_tcl_script -run_dir $fb_uA1_dir
```

## Synthesizing Individual FPGAs Manually

To work on each FPGA manually, follow these steps:

1. Generate individual FPGA partitions with `run system_generate`. See [Generating FPGAs, on page 416](#).

This command generates Tcl scripts for implementing the individual FPGAs. The scripts include commands to compile, pre-map, and map the FPGAs. Each FPGA script also includes the `export vivado` command to generate the files needed for place and route.

2. Set the tool to synthesis mode with the option `set design_flow synthesis` command.
3. Open the database for an individual FPGA and use the generated design files to compile and map the design.

See [The Basic Implementation Flow, on page 472](#), for details.

4. Run place and route for individual FPGAs.
  - Export the files for place and route with the `export vivado` command. You can now run place and route with the `launch vivado` command.
  - To use the default Vivado Tcl script for place and route, run Vivado with commands like the following example, where `myVivadoDir` is the directory generated after running `export vivado`. This directory includes the default script.

```
set vivado_tcl_script [export vivado -path myVivadoDir]
launch vivado $vivado_tcl_script -run_dir $fb_uA1_dir
```

- To use a custom place-and-route script, specify it instead of using the Tcl script generated by `export vivado`, as shown here:

```
launch vivado /home/my_custom_vivado_script.tcl -run_dir $fb_uA1_dir
```

See [Customizing Scripts to Synthesize, Place, and Route FPGAs, on page 430](#) for more information about using customized scripts for individual FPGAs.

## Customizing Scripts to Synthesize, Place, and Route FPGAs

The run system\_generate command creates individual synthesis scripts for each FPGA based on the template in `installDir/lib/protocompiler/sygen_template.tcl`. You can customize these instructions by either editing the FPGA scripts directly, or setting options from the command line when you launch the ProtoCompiler software to run synthesis. You can also customize the place-and-route options by creating a custom file to override the default settings.

- [Customizing Synthesis Scripts](#), on page 430
- [Using Customized Scripts to Run Synthesis and Vivado \(Multi-FPGA\)](#), on page 431
- [Tcl and Environment Variables in Custom Scripts](#), on page 433
- [Customizing Scripts to Run Vivado \(Single-FPGA Designs\)](#), on page 583

### Customizing Synthesis Scripts

The customized scripts to synthesize individual FPGAs can include Tcl variables as well as environment variables.

1. Create the synthesis Tcl scripts by generating partitions with run system\_generate.  
See [Synthesizing Individual FPGAs Using a Script](#), on page 426 for details. Separate scripts are generated for each FPGA, and are customizable.
2. To customize an individual script directly, open the Tcl script file you need, and edit the appropriate line in the Tcl file.

For example, to enable place and route for fb1\_uA, open the Tcl file generated for that FPGA and edit this line:

```
set_if_unset ENABLE_PAR 1
```

The customized scripts can include Tcl variables and environment variables. See [Tcl and Environment Variables in Custom Scripts](#), on page 433 for a list of variables that can be set.

3. To customize synthesis scripts from the command line, specify options to the launch protocompiler command when you synthesize an individual FPGA, using the `--tclcmd` argument.

- Use the launch protocompiler -tclcmd command to specify Tcl code to execute before launching the subsequent script. This can include options you want to set for that FPGA. For example, you can specify sub-hierarchical EDIF files for Vivado. Then define the FPGA Tcl script to which it should apply with a subsequent -script argument. The -tclcmd argument applies to all subsequent scripts, until another -tclcmd argument is specified.

```
launch protocompiler -tclcmd "set ENABLE_PAR1"
 -script synthesis_files/fb1_uA/fb1_uA_srs.tcl
```

The options (-tclcmd) must precede the scripts to which they apply (-script).

- Specify multiple options by separating them with semi-colons and surrounding them with curly braces:

```
launch protocompiler -tclcmd {set ENABLE_BACKANNOTATION 1; set
 ENABLE_PAR 1} -script $scriptA
```

- Specify multiple scripts with separate -script arguments. The -tclcmd options apply to all subsequent scripts. For example, this command runs place and route in both \$scriptA and \$scriptB:

```
launch protocompiler -tclcmd {set ENABLE_PAR 1} -script $scriptA
 -script $scriptB
```

- To specify different settings for different scripts, use multiple sets of -tclcmd and -script arguments. This example runs place and route in fb1\_uA (\$scriptA) and runs both place and route and backannotation in fb1\_uB (\$scriptB):

```
launch protocompiler -tclcmd {set ENABLE_PAR 1} -script $scriptA
 -tclcmd {set ENABLE_PAR 1; set ENABLE_BACKANNOTATION 1}
 -script $scriptB
```

## Using Customized Scripts to Run Synthesis and Vivado (Multi-FPGA)

This procedure describes how to use customized scripts to direct synthesis or place and route for individual FPGAs in a multi-FPGA design.

1. Partition the design and generate the individual FPGAs with run system\_generate. For example:

```
run system_generate -out g0 -path synthesis_files
```

This command creates the partitions at the specified location, with a directory for each FPGA partition:

```
synthesis_files/fb1_uA
synthesis_files/fb1_uB ...
```

Each FPGA directory contains a Tcl script to automate the synthesis of that FPGA:

```
synthesis_files/fb1_uA/fb1_uA_srs.tcl
synthesis_files/fb1_uB/fb1_uV_srs.tcl ...
```

2. To set custom synthesis options, customize the individual Tcl scripts as needed, either directly in the script or from the command line when you start synthesis.
  - Start with the template in \$BUILD/lib/protocompiler/sysgen\_template.tcl.
  - You can further customize the behavior with command line arguments to the launch protocompiler command.

See [Customizing Synthesis Scripts, on page 430](#) for details.

After the compile and map stages finish, the script generates a run\_vivado\_haps.tcl file on a per-FPGA basis, which is used to run place and route with Vivado. The next steps describe how to override the Vivado settings with custom settings.

3. Create a Tcl file with the place-and-route settings you want to run Vivado.

The settings in this custom options file override the settings in the run\_vivado\_haps.tcl file. For example, you can generate a Verilog netlist after place and route or decide that you do not want to generate a bit file:

```
set write_post_par_verilog "1"
set write_bitstream_enable "0"
```

4. Run the scripts for the FPGAs with the launch protocompiler command and the customized Vivado script.
  - For custom PAR explorer scripts, make sure to specify set CUSTOM\_PAR:

```
launch protocompiler -tclcmd {set ENABLE_PAR 1; set
CUSTOM_PAR ../../custom_export_vivado.tcl}
-script ./synthesis_files/fb1_uA/fb1_uA_srs.tcl
```

- For other custom place-and-route scripts, use set VIVADO\_OPTION\_FILE:

```
launch protocompiler -tclcmd {set ENABLE_PAR 1; set
VIVADO_OPTION_FILE ../../custom_export_vivado.tcl}
-script ./synthesis_files/fb1_uA/fb1_uA_srs.tcl
```

Note that the custom script is located two levels up relative to the Tcl files. For information about using the -tclcmd option, see [Customizing Synthesis Scripts, on page 430](#).

## Using Vivado Flags and Options in Run Vivado HAPS Tcl Script

You can use Vivado flags and options in custom run\_vivado\_haps.tcl scripts during the initial exploration of a design to reduce the runtime. Most of these switches are reported by the Vivado in the log. These switches help in implementing a script-based flow without adding the switches to individual Vivado runs.

For example, you can add the Vivado switch route.enableHoldExpnBailout in a custom script run\_vivado\_haps.tcl:

```
catch {set_param route.enableHoldExpnBailout 1}
```

For details on the Vivado flags and options, see the Vivado documentation.

## Tcl and Environment Variables in Custom Scripts

You can add the following Tcl and environment variables in customizable scripts:

| TCL Variable          | Defaults               | To Enable Variable...                                |
|-----------------------|------------------------|------------------------------------------------------|
| ENABLE_PRE_INSTRUMENT | Value: 0<br>State: p0  | {set ENABLE_PRE_INSTRUMENT 1}<br>Runs pre_instrument |
| ENABLE_COMPILE        | Value: 1<br>State: c0  | {set ENABLE_COMPILE 1}<br>Runs compilation           |
| ENABLE_PRE_MAP        | Value: 1<br>State: pm0 | {set ENABLE_PRE_MAP 1}<br>Runs pre-map.              |
| ENABLE_MAP            | Value: 1<br>State: m0  | {set ENABLE_MAP 1}<br>Runs map.                      |

| TCL Variable                     | Defaults                                | To Enable Variable...                                                                                                                             |
|----------------------------------|-----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| ENABLE_EXPORT_VIVADO             | Value: 1                                | {set ENABLE_EXPORT_VIVADO 1}<br>Allows additional TCL script to be sourced with run_vivado_haps.tcl script.                                       |
| ENABLE_PAR                       | Value: 0                                | {set ENABLE_PAR 1}<br>Enables Vivado place and route.                                                                                             |
| ENABLE_PAR_EXPLORER              | Value: 0                                | {set ENABLE_PAR_EXPLORER 1}<br>Enables exploratory place and route.                                                                               |
| ENABLE_BACKANNOTATE              | Value: 0                                | {set ENABLE_BACKANNOTATE 1}<br>Enables backannotation.                                                                                            |
| VIVADO_OPTION_FILE               | -                                       | {set VIVADO_OPTION_FILE<br>../../additional_vivado_option.tcl}<br>Optional file to source in run_vivado_haps.tcl. Used with ENABLE_EXPORT_VIVADO. |
| USER_RTL_IDC                     | -                                       | {set USER_RTL_IDC project_user.idc}<br>For single-FPGA debug.                                                                                     |
| USER_NETLIST_IDC                 | -                                       | {set USER_NETLIST_IDC<br>project_netlist.idc}<br>For single FPGA debug.                                                                           |
| PAR_EXPLORER_SCRIPT              | -                                       | {set PAR_EXPLORER_SCRIPT par_exp.tcl}<br>Additional TCL script to source when running exploratory place and route.                                |
| Read-only Tcl Variable           |                                         | Setting                                                                                                                                           |
| VIVADO_DIR                       | Value:<br>.vivado_<RTL><br>INPUT_FORMAT | Defaults to RTL_INPUT_FORMAT,<br>either HDL or SRS.<br>(Cannot be set from launch<br>protocompiler)                                               |
| Environment Variables            | Default                                 | Setting                                                                                                                                           |
| VM_NETLIST_DIR                   | -                                       | setenv VM_NETLIST_DIR ./Netlists<br>Use with export netlist -path.                                                                                |
| VIVADO_ENABLE_MULTI_MACHINE_MPFS | Value: 0                                | setenv<br>(VIVADO_ENABLE_MULTI_MACHINE_MPFS)<br>Enables multi processing flow (MPF) in a multi-machine scenario.                                  |

| TCL Variable                           | Defaults | To Enable Variable...                                                                                                                                                                                          |
|----------------------------------------|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| VIVADO_REMOTE_MPFSU<br>BIT             | -        | <p>setenv<br/>(VIVADO_REMOTE_MPFSU)<br/>Configures the end-user environment for multi-machine MPF.</p> <p>Example:<br/>setenv(VIVADO_REMOTE_MPFSU)<br/>"qsub -P bnormal -l<br/>mem_free=64G -pe mt 4 -cwd"</p> |
| VIVADO_REMOTE_MPFKIL<br>L              | -        | <p>setenv (VIVADO_REMOTE_MPFKILL)<br/>Stops the specified job in a multi-machine MPF.</p> <p>Example:<br/>setenv (VIVADO_REMOTE_MPFKILL)<br/>"qdel"</p>                                                        |
| VIVADO_ENABLE_SINGLE_<br>MACHINE_MPFSU | Value: 0 | <p>setenv<br/>(VIVADO_ENABLE_SINGLE_MACHINE_MPFSU)<br/>Enables multi processing flow in a single machine scenario.</p>                                                                                         |

## Updating the Top Level with FPGA Implementation Results

You can update the top level with results from various database states at the FPGA level, either because the FPGAs have more accurate values or because they are more current.

You use the `database apply_state` command, as described in the following procedure, to update the top level with area estimates or a mapped FPGA. See [database, on page 30](#) in the *Command Reference* for command syntax details.

1. Partition your design, generate FPGA partitions and implement them as usual.

For individual FPGAs, run the commands you need to generate the results you want. You can then use the `database apply_state` command to update existing database states with the new results. See the following steps for details.

2. Do the following to backannotate area estimates to the top-level design:
  - Pick the critical module or modules for which you want to estimate the area.

- To estimate the area of a sub-module, use the `run compile -top_module` command on the submodule. Then execute the `run pre-map` and `run map` commands to generate an area estimate file (.est).
  - Export the area estimate file for the FPGA with the `export file estimationFile.est` command.
  - Open a pre-partitioned database state for the top level.
  - Specify the database `apply_state -update_est` command with the .est file from submodule, to update the top level with the more accurate area results from the submodule. The following example updates the top level with the results from mod1, mod2, and mod3:

```
database apply_state -update_est {mod1.est mod2.est mod3.est}
```
  - Partition the top-level design with the `run partition` command.
3. To update the top level with another version of a mapped FPGA, follow the steps below.

From the top level, the tool automatically links to the default database state. Use this procedure if you have customized the default name of the FPGA mapped database state, or if you want to change the link to an alternative mapped state.

- Generate a mapped database state by running the `run compile`, `run pre-map`, and `run map` commands on the individual FPGA.
- Open a system generate database state for the top level.
- Specify the database `apply_state -link_module` command with the name of the mapped module and the name of the state. Note the bar between the path and the name of the mapped state:

```
database apply_state -link_module moduleName pathToModule|stateName
```

This command links the specified mapped state to the top-level design.

- You can now rerun `run system_generate` at the top level.
4. At the individual FPGA level, use the `database apply_state -import_vivado` command to import place-and-route results back into the FPGA. See [Importing Place and Route Results for Backannotation](#), on page 584 for details.

# Using Multi-Design Mode (MDM)

Multi-design mode (MDM) lets you pool your hardware resources in HAPS farms and share the same HAPS systems between multiple users or designs. You can run multiple designs concurrently, either locally or remotely. This mode is also called multi-use or multi-user mode. Use multi-design mode to prototype designs or to validate them.

Multi-design mode is not limited to single FPGAs, but can be spread across multiple systems. For example, a 5-FPGA design can be implemented on FPGAs A, B, C, and D of system 1 and FPGA A of system 2, while another 3-FPGA design can be implemented on the remaining FPGAs (B, C, and D) of the system 2.

You can use mixed system designs for multi-design mode.

For details, see the following:

- [Running Designs in Multi-Design Mode: HAPS-100](#), on page 437
- [Using UMRBus 3.0 with HAPS-100 Modules](#), on page 320
- [Running MDM in a Mixed-Design Setup: HAPS-100 and HAPS-80](#), on page 442
- [Running MDM in a Mixed-Design Setup: HAPS-80 and HAPS-70](#), on page 444
- [MDM Planning Guidelines for HAPS-70 and HAPS-80](#), on page 446
- [Instantiating CAPIM\\_UI](#), on page 452

## Running Designs in Multi-Design Mode: HAPS-100

This procedure describes how to set up designs for multi-design mode with HAPS-100 systems. The following procedure uses an example to illustrate the methodology. The example is based on a single HAPS-100 system design which is partitioned into two FPGAs. The design has fixed interconnects between two FPGAs, and the buffer type is DTD (`haps100_DTD_builtin`).

In HAPS-100 systems, the fixed connectors are between FPGAs A and B, and between C and D. The User-1 design targets FPGAs A and B and the User-2 design uses FPGAs C and D. Although you can use MDM to run multiple

designs on shared resources, for this example the same bit files from A and B are configured on C and D. Then two runtime sessions are used to debug the same design, configured as User-1 and User-2.

### 1. Export files for runtime.

- Make sure you have set up the appropriate tools and hardware.
- In protocompiler100, run the entire software flow through place-and-route.
- Export the bit files to the runtime directory as usual:

```
export runtime -path export_results
```

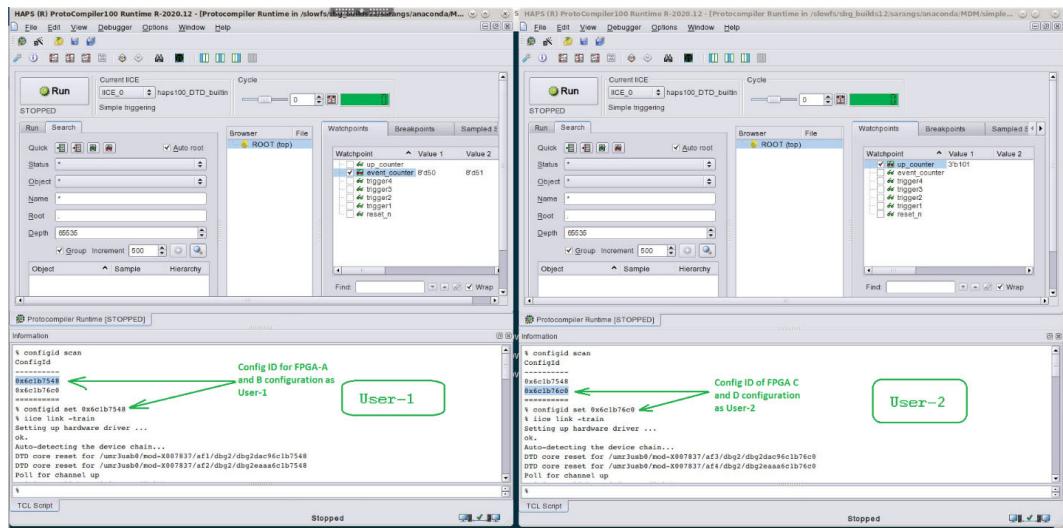
This `export_results` directory contains bit files, the debug project (`debug.prj`) and the TSD (target system definition) file, `targetsyste.tsd`.

### 2. Use the TSD file to configure the HAPS-100 system.

- Create a second copy of the `targetsyste.tsd` file for User-2, because the same design is being used by both. In the second file, replace A and B with C and D to ensure that the board settings are correct for C and D.

The following figure shows the TSD files for User-1 and User-2, with User-1 using FPGAs A and B, and User-2 using FPGAs C and D. In this scenario, the bit files and the FPGA IDs of User-2 are same as User-1.

```
targetsyste.tsd (/slowfs/sbg_bu...2fpga/export_results/system) - GVIM
File Edit Tools Syntax Buffers Window Help
1 Board_system_create -haps -name Test_board -tsd
2 board_system_create -add HAPS100_4F -name FBI1
3 board_system_configure -top_io { FBI1.A4}
4 board_system_configure -top_io { FBI1.B4}
5 board_system_configure -top_io { FBI1.A5}
6 board_system_configure -top_io { FBI1.B5}
7 board_system_configure -top_io { FBI1.A6}
8 board_system_configure -top_io { FBI1.B6}
9 board_system_configure -clk_src {FBI1.PLL1.CLK1} -frequency 10000 -name d_clk
10 board_system_configure -clock FBI1.UA.CLK1 d_clk
11 board_system_configure -clock FBI1.UB.CLK1 d_clk
12 board_system_configure -voltage FBI1.A 1.20
13 board_system_configure -voltage FBI1.A5 1.20
14 board_system_configure -voltage FBI1.AB 1.20
15 board_system_configure -voltage FBI1.B4 1.20
16 board_system_configure -voltage FBI1.B5 1.20
17 board_system_configure -voltage FBI1.B6 1.20
18 board_system_configure -fpga_id FBI1.UA (0xDAC9)
19 board_system_configure -fpga_file FBI1.UA (.../FBI1_uA/FBI1_uA.bit)
20 board_system_configure -fpga_id FBI1.UB (0xEAAA)
21 board_system_configure -fpga_file FBI1.UB (.../FBI1_uB/FBI1_uB.bit)
~ Already at oldest change
1,1 All Already at oldest change
targetsyste2.tsd (/slowfs/sbg_bu...2fpga/export_results/system) - GVIM1
File Edit Tools Syntax Buffers Window Help
1 Board_system_create -haps -name Test_board -tsd
2 board_system_create -add HAPS100_4F -name FBI1
3 board_system_configure -top_io { FBI1.C4}
4 board_system_configure -top_io { FBI1.D4}
5 board_system_configure -top_io { FBI1.C5}
6 board_system_configure -top_io { FBI1.D5}
7 board_system_configure -top_io { FBI1.C6}
8 board_system_configure -top_io { FBI1.D6}
9 board_system_configure -clk_src {FBI1.PLL1.CLK1} -frequency 10000 -name d_clk
10 board_system_configure -clock FBI1.UC.CLK1 d_clk
11 board_system_configure -clock FBI1.UD.CLK1 d_clk
12 board_system_configure -voltage FBI1.C 1.20
13 board_system_configure -voltage FBI1.C5 1.20
14 board_system_configure -voltage FBI1.C6 1.20
15 board_system_configure -voltage FBI1.D 1.20
16 board_system_configure -voltage FBI1.D5 1.20
17 board_system_configure -voltage FBI1.D6 1.20
18 board_system_configure -fpga_id FBI1.UC (0xDAC9)
19 board_system_configure -fpga_file FBI1.UC (.../FBI1_uA/FBI1_uA.bit)
20 board_system_configure -fpga_id FBI1.UD (0xEAAA)
21 board_system_configure -fpga_file FBI1.UD (.../FBI1_uB/FBI1_uB.bit)
~ Already at oldest change
1,1 All Already at oldest change
```



### 3. Configure the FPGAs.

- Use Confpro to clear the HAPS-100 system.
- Configure the FPGAs individually as User-1 (FPGAs A and B) and User-2 (FPGAs C and D) with the `cfg_project_configure` command. Use the `-noclear` option with the command to ensure that reconfiguring does not clear the FPGAs of the other user.

The figure below shows scripts used to configure the system individually for two users.

The image shows two GVIM editors side-by-side. Both editors have a top toolbar with 'File', 'Edit', 'Tools', 'Syntax', 'Buffers', 'Window', and 'Help'.

**tcl.conf1.tcl Content:**

```
26 puts "\n"
27 puts "Cleaning system"
28 #cfg_project_clear_shandle
29 puts "Configuring Hardware DM-10 from generated targetsystem.tsd ..."
30 if {[catch {cfg_project_configure shandle export_results/system/targetsystem.tsd -noclear} h]} {
31 puts "\n\\033[1;31mError during programming DM-10 Hardware..\\l \\033\\0m"
32 puts "\\033[1;31m\\033\\0m"
33 } else {
34 puts "\\n\\033[1;32mConfigured Hardware DM-10 successfully..\\l \\033\\0m"
35 }
36 puts "Applying Hardware reset..."
37 cfg_reset_set shandle FB1.UA 0
38 after 2000
39 cfg_reset_set shandle FB1.UA 1
40 puts "Unclosing handle shandle"
41 clsh_close shandle
42 after 3000
43 puts *****
44 exit
search hit BOTTOM, continuing at TOP
```

**tcl.conf2.tcl Content:**

```
26 puts "\n"
27 puts "Cleaning system"
28 #cfg_project_clear_shandle
29 puts "Configuring Hardware DM-10 from generated targetsystem.tsd ..."
30 if {[catch {cfg_project_configure shandle export_results/system/targetsystem2.tsd -noclear} h]} {
31 puts "\n\\033[1;31mError during programming DM-10 Hardware..\\l \\033\\0m"
32 puts "\\033[1;31m\\033\\0m"
33 } else {
34 puts "\\n\\033[1;32mConfigured Hardware DM-10 successfully..\\l \\033\\0m"
35 }
36 puts "Applying Hardware reset..."
37 cfg_reset_set shandle FB1.UA 0
38 after 2000
39 cfg_reset_set shandle FB1.UA 1
40 puts "Unclosing handle shandle"
41 clsh_close shandle
42 after 3000
43 puts *****
44 exit
search hit BOTTOM, continuing at TOP
```

4. After the hardware is configured, use the `lsumr3` command to check that the correct CAPIMs are listed for each FPGA.

\$BUILD/bin/lsumr3

| DEVICE    | ADDRESS | HID | CONNECTED | PATH                                            |
|-----------|---------|-----|-----------|-------------------------------------------------|
| /umr3usb0 | 0x20    | 0x0 | 0         | /umr3usb0/mod-X007837/uf1/sys/sys_dac96c1b7548  |
| /umr3usb0 | 0x30    | 0x0 | 0         | /umr3usb0/mod-X007837/uf1/usr/dbg5dac96c1b7548  |
| /umr3usb0 | 0x48    | 0x0 | 0         | /umr3usb0/mod-X007837/af1/dbg0/dbg0dac96c1b7548 |
| /umr3usb0 | 0x68    | 0x0 | 0         | /umr3usb0/mod-X007837/af1/dbg2/dbg2dac96c1b7548 |
| /umr3usb0 | 0x58    | 0x0 | 0         | /umr3usb0/mod-X007837/af1/dbg1/dbg1dac96c1b7548 |
| /umr3usb0 | 0x78    | 0x0 | 0         | /umr3usb0/mod-X007837/af1/sys/hsm_a80001111     |
| /umr3usb0 | 0x24    | 0x0 | 0         | /umr3usb0/mod-X007837/uf2/sys/sys_eaaa6c1b7548  |
| /umr3usb0 | 0x34    | 0x0 | 0         | /umr3usb0/mod-X007837/uf2/usr/dbg5aaa6c1b7548   |
| /umr3usb0 | 0x4c    | 0x0 | 0         | /umr3usb0/mod-X007837/af2/dbg0/dbg0eaaa6c1b7548 |
| /umr3usb0 | 0x6c    | 0x0 | 0         | /umr3usb0/mod-X007837/af2/dbg2/dbg2eaaa6c1b7548 |
| /umr3usb0 | 0x5c    | 0x0 | 0         | /umr3usb0/mod-X007837/af2/dbg1/dbg1eaaa6c1b7548 |
| /umr3usb0 | 0x7c    | 0x0 | 0         | /umr3usb0/mod-X007837/af2/sys/hsm_a80001211     |
| /umr3usb0 | 0x22    | 0x0 | 0         | /umr3usb0/mod-X007837/uf3/sys/sys_dac96c1b76c0  |
| /umr3usb0 | 0x32    | 0x0 | 0         | /umr3usb0/mod-X007837/uf3/usr/dbg5dac96c1b76c0  |
| /umr3usb0 | 0x4a    | 0x0 | 0         | /umr3usb0/mod-X007837/af3/dbg0/dbg0dac96c1b76c0 |
| /umr3usb0 | 0x6a    | 0x0 | 0         | /umr3usb0/mod-X007837/af3/dbg2/dbg2dac96c1b76c0 |
| /umr3usb0 | 0x5a    | 0x0 | 0         | /umr3usb0/mod-X007837/af3/dbg1/dbg1dac96c1b76c0 |
| /umr3usb0 | 0x7a    | 0x0 | 0         | /umr3usb0/mod-X007837/af3/sys/hsm_a80001311     |
| /umr3usb0 | 0x26    | 0x0 | 0         | /umr3usb0/mod-X007837/uf4/sys/sys_eaaa6c1b76c0  |
| /umr3usb0 | 0x36    | 0x0 | 0         | /umr3usb0/mod-X007837/uf4/usr/dbg5aaa6c1b76c0   |
| /umr3usb0 | 0x4e    | 0x0 | 0         | /umr3usb0/mod-X007837/af4/dbg0/dbg0eaaa6c1b76c0 |
| /umr3usb0 | 0x6e    | 0x0 | 0         | /umr3usb0/mod-X007837/af4/dbg2/dbg2eaaa6c1b76c0 |
| /umr3usb0 | 0x5e    | 0x0 | 0         | /umr3usb0/mod-X007837/af4/dbg1/dbg1eaaa6c1b76c0 |
| /umr3usb0 | 0x7e    | 0x0 | 0         | /umr3usb0/mod-X007837/af4/sys/hsm_a80001411     |
| /umr3usb0 | 0x11    | 0x0 | 0         | /umr3usb0/mod-X007837/cf1/hsm_c00001110         |
| /umr3usb0 | 0x21    | 0x0 | 0         | /umr3usb0/mod-X007837/cf1/hsm_d00001110         |
| /umr3usb0 | 0x19    | 0x0 | 0         | /umr3usb0/mod-X007837/cf2/hsm_c00001210         |
| /umr3usb0 | 0x29    | 0x0 | 0         | /umr3usb0/mod-X007837/cf2/hsm_d00001210         |
| /umr3usb0 | 0x15    | 0x0 | 0         | /umr3usb0/mod-X007837/cf3/hsm_c00001310         |
| /umr3usb0 | 0x25    | 0x0 | 0         | /umr3usb0/mod-X007837/cf3/hsm_d00001310         |
| /umr3usb0 | 0x1d    | 0x0 | 0         | /umr3usb0/mod-X007837/cf4/hsm_c00001410         |
| /umr3usb0 | 0x2d    | 0x0 | 0         | /umr3usb0/mod-X007837/cf4/hsm_d00001410         |
| /umr3usb0 | 0x7     | 0x0 | 0         | /umr3usb0/mod-X007837/hsm_s00001010             |
| /umr3usb0 | 0xb     | 0x0 | 0         | /umr3usb0/mod-X007837/hsm_d00001010             |
| /umr3usb0 | 0xf     | 0x0 | 0         | /umr3usb0/mod-X007837/hsm_m00001010             |

This command shows all the CAPIMs on the hardware after successful configuration. This is how to interpret the output from lsumr3:

- There is information for individual FPGAs:

|         |               |        |
|---------|---------------|--------|
| uf1/af1 | FPGAs A and B | User-1 |
| uf1/af2 |               |        |
| uf3/af3 | FPGAs C and D | User-2 |
| uf4/af4 |               |        |

- Check the FPGA ID on system CAPIM to ensure that the bit files are correctly configured on the targeted FPGAs. The FPGA ID is the first four characters in the system CAPIM. For example, in the system CAPIM /uf1/sys/sys\_dac96c1b7548, the first 4 characters are dac9, so dac9 is the FPGA ID for FPGA A.
- Also check the configuration ID after successful configuration. The ID is updated every time the design is re-configured. The post-configuration ID is the last 8 characters in the system CAPIM. For example, in the system CAPIM sys\_dac96c1b7548, the last 8

characters are 6c1b7548, so this is the ID after configuration. This ID is then used to run user-based debug.

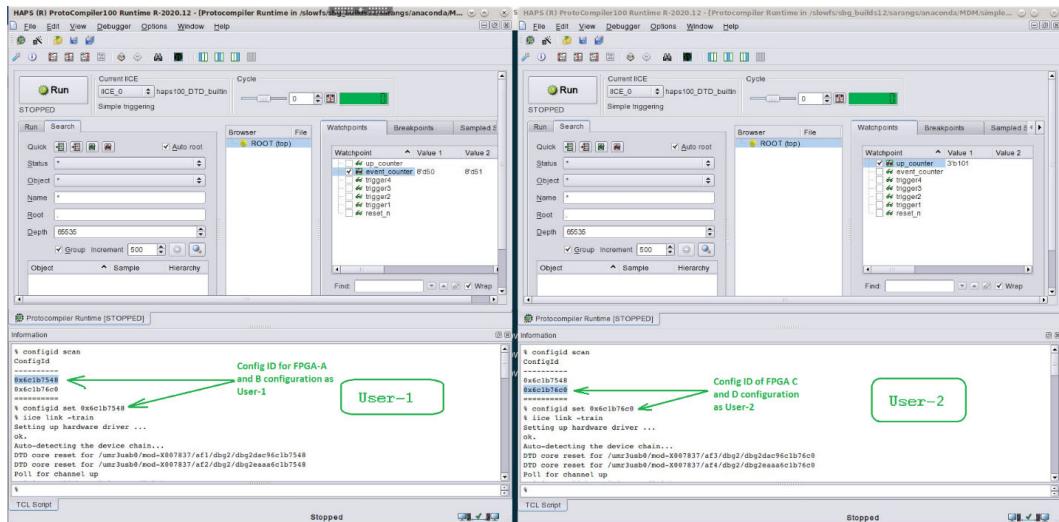
5. Open the debug project to run debug independently.
  - Each user must open protocompiler100\_runtime separately.
  - Each user can then open the same debug project in the runtime directory that was exported in step 1.
6. Use the configuration IDs to run individual debug sessions.
  - In the Runtime TCL shell, set the configuration ID from the lsumr3 output, as described previously. Use the configid set *config\_id* command.

Each user specifies the ID for their FPGAs. For example, User-1 would use 6c1b7548 to run debug on FPGAs A and B, but not C and D.

  - Instead of using lsumr3 to get the ID, users can run this sequence of commands in the runtime Tcl shell to get and set the ID.

```
configid scan
configid set config_id
```

The following figure shows IDs set by User-1 and User-2 in their respective runtime Tcl shell windows.



7. After the IDs have been successfully set, run link training.

- Enter the `iice link -train` command in the runtime Tcl shell to run link training.
- Check the user-specific link training message to ensure that the link training was successful on the correct FPGA. Here, `af1` and `af2` are used for FPGAs A and B; `af3` and `af4` are used for FPGAs C and D.

User-1 Runtime Shell

```
=====
Setting up hardware driver ...
ok.
Auto-detecting the device chain...
DTD core reset for /umr3usb0/mod-X007837/af1/dbg2/dbg2dac96c1b7548
DTD core reset for /umr3usb0/mod-X007837/af2/dbg2/dbg2eaaa6c1b7548
Poll for channel up
Waiting on link training to finish...
Links are successfully trained!
```

User-2 Runtime Shell

```
=====
Setting up hardware driver ...
ok.
Auto-detecting the device chain...
DTD core reset for /umr3usb0/mod-X007837/af3/dbg2/dbg2dac96c1b76c0
DTD core reset for /umr3usb0/mod-X007837/af4/dbg2/dbg2eaaa6c1b76c0
Poll for channel up
Waiting on link training to finish...
Links are successfully trained!
```

- After successful link training, both users can individually debug their design by setting breakpoints and running the debugger.

## Running MDM in a Mixed-Design Setup: HAPS-100 and HAPS-80

This procedure outlines how to set up designs for multi-design mode and how to run MDM with a mixed setup of HAPS-100 and HAPS-80 systems.

1. Make sure you have the appropriate tools and hardware setup.

- Set up each HAPS-100 system to connect to the host computer separately through USB or QSFP, using the UMRBus3 protocol. The HAPS-100 system does not have CDE connectors to connect to HAPS-80.
  - Set up the HAPS-80 systems to connect to the host computer separately from the HAPS-100 systems. Use UMRBus and the UMRBus2 protocol for HAPS-80.
2. Plan the designs.
    - Floorplan where the designs go.
    - Plan the clocks and resets.
  3. Instantiate CAPIM\_UI.
  4. Set up configuration and queuing in Confpro.

Only one Confpro session is available per system. You can use a Confpro script to timeshare successfully.

    - First, configure the system without any user designs. Configure the clocks and voltages.
    - Configure the first user design. Use the `umrbusscan` command to report the addresses and check what is being used by the design.
    - Configure the other designs one at a time. Subsequent `umrbusscan` commands report the addresses for all configured designs.
    - Set up the designs to share time on the UMRBus, with sequential access.
    - Use the Confpro `cfg_open` command with the `-wait` option to avoid lockout messages during system configuration.
  5. Run the designs sequentially.

If you are using the debug flow with MDM on HAPS-80 systems, be careful about using built-in memory for debug. Built-in memory is only available on FPGA A, so use it accordingly for MDM.

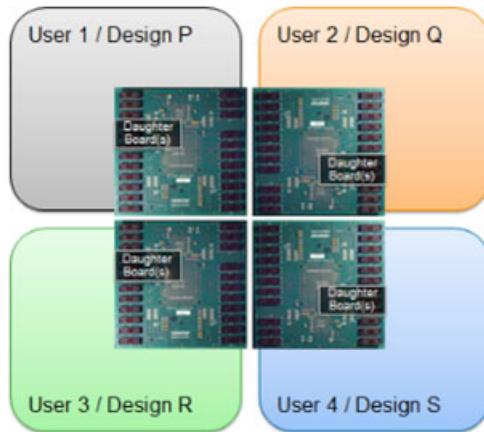
If you are using a transactor design, make sure to follow the guidelines in [Using Transactors in Multi-Design Mode, on page 451](#).

## Running MDM in a Mixed-Design Setup: HAPS-80 and HAPS-70

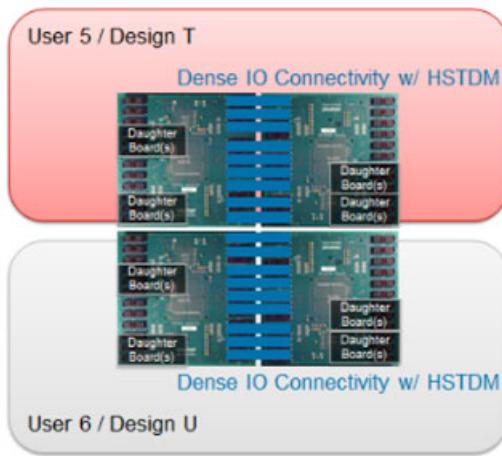
This procedure outlines how to set up designs for multi-design mode and how to run the designs for HAPS-70 and HAPS-80 systems.

1. Make sure you have the appropriate tools and hardware setup. You can have design setups as shown below:

4 single FPGA designs



Two 2-FPGA designs with HSTDMD



Multi-design mode is not limited to single FPGAs, but can be spread across multiple systems. For example, a 5-FPGA design can be implemented on FPGAs A, B, C, and D of one system and FPGA A of

the second system, and another 3-FPGA design can be implemented on the remaining FPGAs (B, C, and D) of the second system.

Multi-design mode does not require cascaded systems. If they are not cascaded, remember that UMRBus GCLK0 will not be synchronized between systems through the CDE cable. If the CDE cable for UMRBus is connected, the systems are chained. The CDE CLK\_LEFT and CLK\_RIGHT are configuration-dependent. See [Chaining Systems for MDM, on page 446](#) for more information.

If the systems are not chained, synchronize the user clocks independently, or use an ECDB to distribute them.

## 2. Plan the designs.

- Floorplan where the designs go. See [MDM Planning Guidelines for HAPS-70 and HAPS-80, on page 446](#) for more information.
- Plan the clocks and resets. See [Working with Design Resets for Multi-Design Mode, on page 449](#) and [Distributing Clocks for Multi-Design Mode, on page 450](#).

## 3. Instantiate CAPIM\_UI.

See [Instantiating CAPIM\\_UI, on page 452](#) for details about instantiating CAPIMs and the addresses you can use for them.

## 4. Set up configuration and queuing in Confpro.

Only one Confpro session is available per system. You can use a Confpro script to timeshare successfully.

- First, configure the system without any user designs. Configure the clocks and voltages.
- Configure the first user design. Use the umrbusscan command to report the addresses and check what is being used by the design.
- Configure the other designs one at a time. Subsequent umrbusscan commands report the addresses for all configured designs.
- Set up the designs to share time on the UMRBus, with sequential access.
- Use the Confpro -wait option to avoid lockout messages during system configuration. For example:

```
cfg_open haps80 emu:8 -wait
cfg_config_data cfg0 FB1_A myDesigns/top.bit
cfg_open haps80 emu:8 -wait
cfg_config_data cfg0 FB1_B myDesigns/FB1_uA/FB1_uA.bit
```

5. Run the designs sequentially.

If you are using the debug flow with MDM on HAPS-80 systems, be careful about using built-in memory for debug. Built-in memory is only available on FPGA A, so use it accordingly for MDM.

If you are using a transactor design, make sure to follow the guidelines in [Using Transactors in Multi-Design Mode, on page 451](#).

## MDM Planning Guidelines for HAPS-70 and HAPS-80

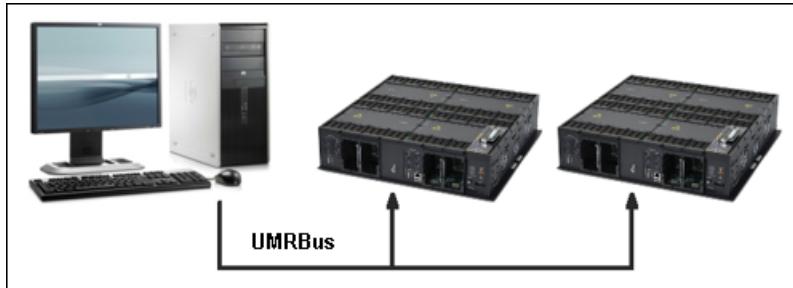
For multi-design mode, the designs need to be able to share and utilize the infrastructure of the HAPS systems. You must plan how design resets, design clocks, design UMRBus and CAPIMs will be shared; and how configuration and queuing is set up. See these topics for details:

- [Chaining Systems for MDM, on page 446](#)
- [Guidelines for Floorplanning MDM Designs, on page 447](#)
- [Working with Design Resets for Multi-Design Mode, on page 449](#)
- [Distributing Clocks for Multi-Design Mode, on page 450](#)
- [Using Transactors in Multi-Design Mode, on page 451](#)
- [Instantiating CAPIM\\_UI, on page 452](#)

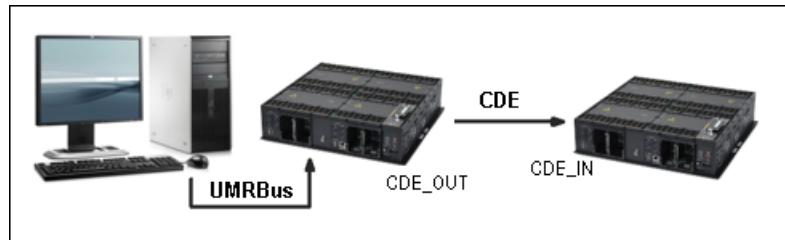
### Chaining Systems for MDM

The UMRBus chain defines the system for MDM. A single HAPS system is defined as a chained system on a single physical UMRBus.

- A chained system shares the same UMRBus resources (0 to 32). This means that two HAPS-80 S104 systems with separate UMRBus links are considered separate systems, even if they share the same host PC.



- A CDE cable connection in a chained system is considered a hard connection, because it is independent of the configuration setup. Two HAPS-80 S104 systems connected by CDE cable are treated as one HAPS-80 S208 system:

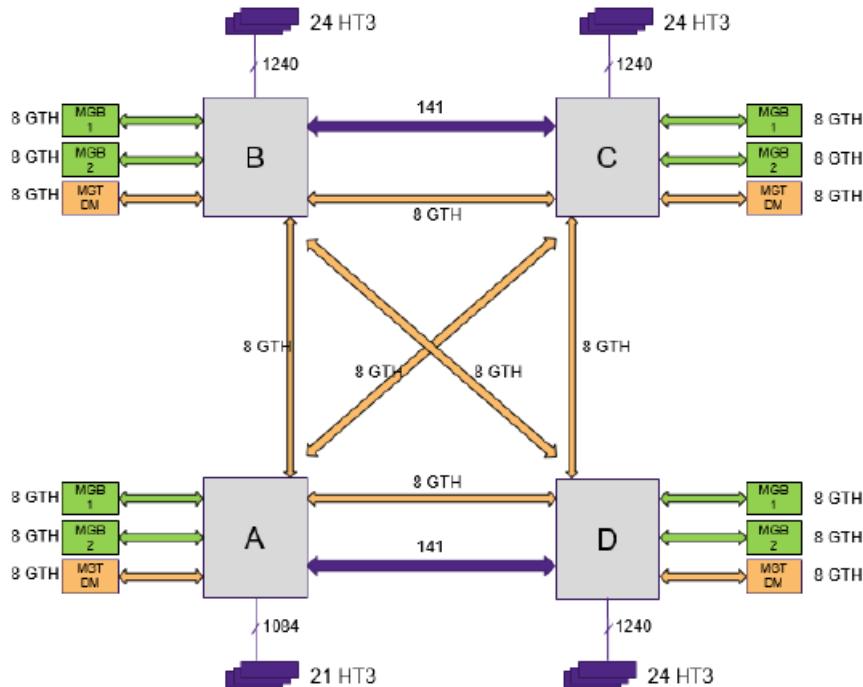


- GCLK0 is not synchronized through the CDE cable. If you need to synchronize it, use the clock connector definitions for the CDE to define soft connections that are configuration-dependent. You can also use a HAPS-ECDB to distribute the user clocks.
- One Confpro session is available per system, with each design on the system time-sharing the UMRBus, and accessing it sequentially.
- Initiate HSTDMD training from the Confpro GUI.
- You can configure a HAPS system without an FPGA binary, and add it in later. You can also configure one FPGA while you are using another.

## Guidelines for Floorplanning MDM Designs

- Each design can occupy any number of FPGAs. For example, a HAPS-80 S208 system supports multi-design combinations like the following: eight 1-FPGA designs; one 5-FPGA design and one 3-FPGA design; one 6-FPGA design and two 1-FPGA designs; and so on.

- Designs can spread across different HAPS systems. A single 5-FPGA design can be implemented on FPGAs A, B, C, and D of a HAPS-80 S104 and FPGA A of a second HAPS-80 S104 system.
- Multiple designs cannot share FPGAs or daughter boards.
- The designs can be the same design or unique.
- Multiple designs must share the same HAPS system infrastructure, like global clocks, design clocks and reset, UMRBus, and CAPIMs.



- For HAPS-80 and HAPS-100 designs, take advantage of the built-in interconnect architecture. The figure above shows the interconnects of the HAPS-80 S104. For two 2-FPGA designs on a HAPS-80 S104 for example, use FPGAs A and D for one design, and B and C for the other.
- Leverage the built-in features of the HAPS-80 architecture when planning for debug with MDM designs. Refer to the preceding figure.
  - To debug designs on individual FPGAs or pairs, use the built-in resources on FPGA A. See [Running Multi-FPGA Debug with HAPS-80 Built-in Memory, on page 712](#).

- To run single-FPGA debug on all the FPGAs, add additional DDR memory boards as needed. See [Running Single-FPGA Debug on a HAPS-80 FPGA, on page 703](#).
- If you do not want to use additional hardware, use synchronous GSV ([Using Global State Visibility \(GSV\), on page 753](#)). Use BRAM IICE to trigger the clock module. You can also use distributed BRAMs for multiple FPGAs.

## Working with Design Resets for Multi-Design Mode

For HAPS-70 systems, the global RESETn signal feeds all FPGAs in a one-CDE chain, or all four FPGAs of a HAPS-70 S48 system.

HAPS-80 systems are more flexible. Each FPGA has individual global RESETn signals. The signals can be globally or controlled individually for each FPGA through Confpro commands.

### 1. Set up the reset signals.

- For HAPS-80 single-system designs (four or fewer FPGAs), use the local reset generated from the System IP. Each FPGA has its own local reset. You can specify the resets globally or individually for each FPGA with the Confpro cfg\_reset\_set command.

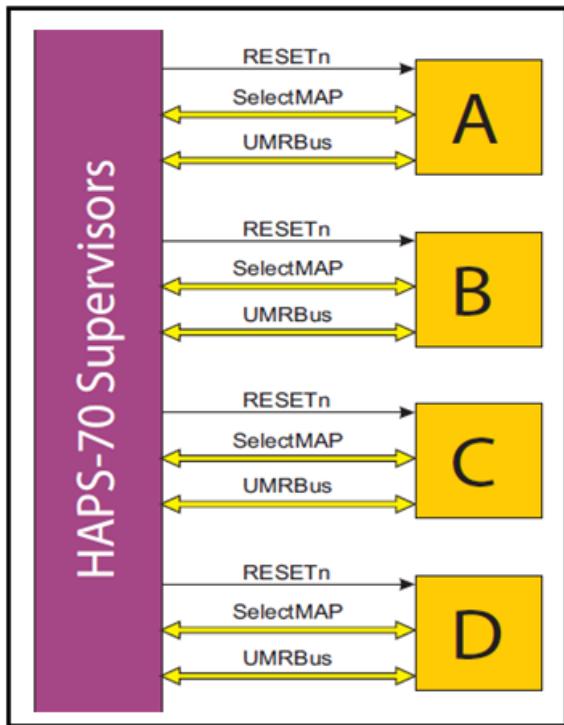
```
cfg_reset_set cfg0 FB1 0
cfg_reset_set cfg0 FB1_B 1
```

- For HAPS-70 single-system designs (four or fewer FPGAs) use a GPIO daughter board or use Confpro to write to the CAPIM.
- For multi-FPGA designs with HSTDMD, use HAPS-controlled training ([Using HAPS-Controlled HSTDMD Training, on page 380](#)). For this case, use the GCLK to drive resets. Resets can only be driven from FPGAs A and B, so make sure that A and B are in separate pairs. The important thing to ensure is that when you use asynchronous resets, the two FPGAs are synchronized when they come out of reset.

### 2. Distribute the reset signals.

- For HAPS-80 systems, distribute the reset signal as described in [Assigning and Distributing HAPS-80 Reset, on page 278](#).
- For HAPS-70 systems, use a global clock instead of the global RESETn signal if you want independent controls for the designs. The RESETn signal feeds all the FPGAs in one CDE chain. On a HAPS-70 S48

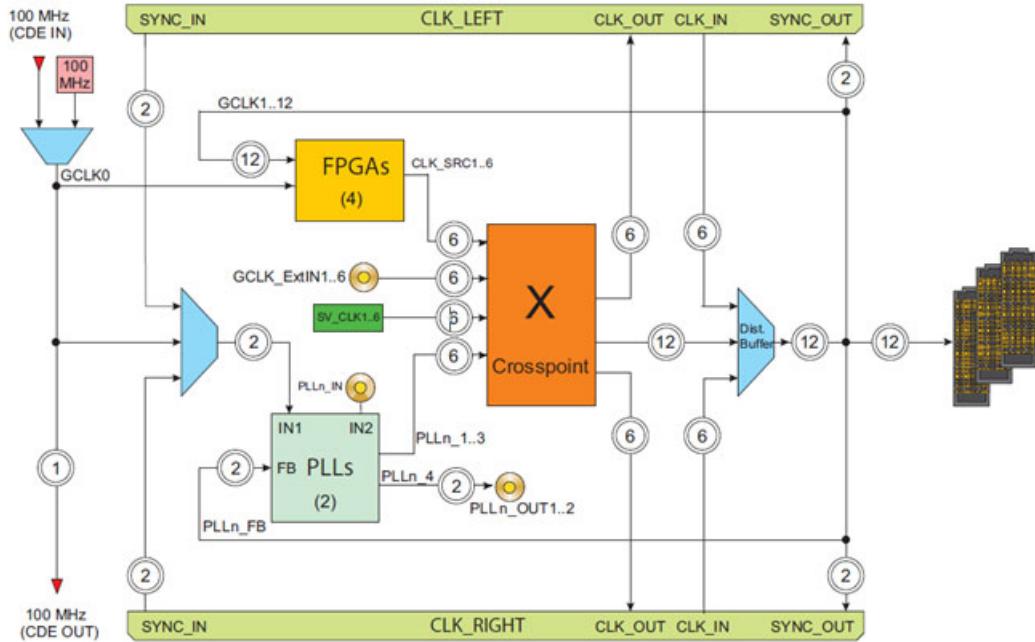
system, the `RESETn` signal feeds all four FPGAs, as shown in the following figure.



## Distributing Clocks for Multi-Design Mode

Follow these guidelines when working with shared clocks in multi-use mode:

- Do not share clocks between designs. HAPS system clocks are common to all the FPGAs, as shown in the figure below. Plan your clock scheme accordingly.



- Avoid using clocks generated by the FPGAs because FPGA-generated clocks feed into the global clocks for the system. Do not use onboard inter-FPGA traces for MDM.

If you must use FPGA-generated clocks, review the implications of your setup before implementing it, and make sure you define it correctly, using FPGAs A or B as clock sources. See [Working with HAPS-80 Resets, on page 278](#) for details.

- If you have multiple copies of the same design configured on a HAPS system in multi-use mode, remember that the clocks are common to all the FPGAs in the system. Changes that affect the global clock setting affect the other FPGAs. So if you make a change, make sure to retain the global clock connections so that the same bit files can be reused. If you do not retain the original global clock connections, you must recompile the design to generate new bit files.

## Using Transactors in Multi-Design Mode

- Regenerate transactors using the latest transactor package. Specify a global define to enable MDM:

```
`define UMR_USE_LOCATION_ID
```

4. Run the design and check the logs.
  - Check the log file after compilation to make sure the CAPIM\_UI is compiled (CG364). The file reports the location IDs of user CAPIMs.
  - After the pre-partition, partition, or system route stages, view the log file to check whether the address was implemented dynamically.

## Instantiating CAPIM\_UI

CAPIM\_UI supports HAPS-70 and HAPS-80 systems. For HAPS-100 modules, refer to [Using UMRBus 3.0 with HAPS-100 Modules, on page 455](#).

The CAPIM\_UI eliminates the need for multiple compile runs to address conflicts caused by multiple copies of the same bit file. CAPIM\_UI uses dynamic addressing instead of static addressing. The dynamic addresses are assigned by the supervisor at run time.

1. Instantiate CAPIM\_UI in your design.

Get the module definition from the /umr\_capim.v file, located in *instal/Dir/lib/vlog*. The CAPIM address must include the FPGA location ID and non-zero MSB bit address. See [CAPIM\\_UI Example, on page 455](#).

CAPIM\_UI is the default for HAPS-80 systems, but not for HAPS-70.

The number of CAPIMs that can be shared in multi-user mode depends on the hardware setup:

| System Size                    | CAPIMs/FPGA                        | Valid CAPIM Addresses |
|--------------------------------|------------------------------------|-----------------------|
| 4 or fewer<br>chained systems  | 7<br>(Including transactor CAPIMs) | 1, 2, 3, 4, 5, 6, 7   |
| More than 4<br>chained systems | 3<br>(Including transactor CAPIMs) | 1, 2, 3               |

Note that the available CAPIM addresses are different from non-MDM mode. In non-MDM mode, 32 CAPIMs can be used for the user design, addresses 1-31.

2. Check that the UMR\_USE\_LOCATION\_ID parameter in the CAPIM definition is set to 1 to use dynamic addressing.

```
capim_ui #(
 .UMR_CAPIM_ADDRESS(1),
 .UMR_CAPIM_TYPE(16'hA001),
 .UMR_DATA_BITWIDTH(UMR_DATA_BITWIDTH),
 .UMR_CAPIM_COMMENT_STRING("I_CAPIM_1"),
 .UMR_USE_LOCATION_ID(1))
```

If it is set to 0, the CAPIM address is defined by UMR\_CAPIM\_ADDRESS.

- Dynamic addressing avoids address conflicts. Dynamic addressing is the default for HAPS-80 systems but not for HAPS-70 systems. When you configure multiple designs with UMRBus, pay special attention to the CAPIM IDs, and make sure that you do not unintentionally run into the other designs and cause address conflicts. Instead of using static addresses, the supervisor sets the CAPIM address dynamically at runtime according to the FPGA where the CAPIM\_UI is partitioned, thus eliminating address conflicts.

The FPGA name is important to identify the FPGA when configuring with Confpro, and for debug. In the table, FPGA names follow the convention **FBX\_Y**, where **X** is the system number in the CDE chain, and **Y** is the FPGA name (A-D).

| FPGAs     | Dynamic FPGA Location ID (XXX) |
|-----------|--------------------------------|
| FB[1-4]_A | 000                            |
| FB[1-4]_B | 001                            |
| FB[1-4]_C | 010                            |
| FB[1-4]_D | 011                            |
| FB[5-8]_A | 100                            |
| FB[5-8]_B | 101                            |
| FB[5-8]_C | 110                            |
| FB[5-8]_D | 111                            |

With dynamic addressing, the UMR\_CAPIM address is handled with **fpga\_location\_id** (based on the FPGA), rather than as a static address.

```
{fpga_location_id[2:0], UMR_CAPIM_ADDRESS [1:0] }
```

- For HAPS-70 systems to use dynamic addressing based on location, you must also set a global attribute in the FDC file. It is not required for HAPS-80 systems.

```
define_global_attribute syn_umr_use_capim_wp 1
```

- For HAPS-70 systems with the older CAPIMs, avoid address conflicts by manually assigning different base addresses for the FPGAs in each idc file. Otherwise you might encounter No instrumented bit file found errors.

fpga\_A.idc file: device capimbaseaddr 10

fpga\_B.idc file: device capimbaseaddr 12

3. Reduce the number of HAPS systems in chain with define UMR\_CHAIN\_LIMIT4 parameter.

The default chain limit is 8, where as user CAPIMs are shared between pairs of HAPS systems (first and fifth; second and sixth, third and seventh; fourth and eighth) with a maximum of 3 user CAPIMs per FPGA.

If you set UMR\_CHAIN\_LIMIT4, the number of HAPS systems in the chain is 4, so the number of user CAPIMs per FPGA increases. There are now maximum 7 user CAPIMs per FPGA.

If the number of user CAPIMs is a problem, consider using separate systems. For example, use two 4-FPGA systems instead of a chained 8-FPGA system. Handle GCLK0, which is synchronized through the CDE.

4. After running pre-partition, check the CAPIM addresses in the report log to make sure that no static addresses are assigned.

The address must show CAPIM\_WP:

| CAPIM   | Bus | FPGA | Width | Addr     | Type     | Comment        |
|---------|-----|------|-------|----------|----------|----------------|
| I_capim | -   | -    | 8     | CAPIM_WP | CAPIM_WP | I_gpio_master  |
| I_capim | -   | -    | 8     | CAPIM_WP | CAPIM_WP | I_axi4_master2 |
| I_capim | -   | -    | 8     | CAPIM_WP | CAPIM_WP | I_axi4_master1 |

## CAPIM\_UI Example

```
capim_ui #(
 .UMR_CAPIM_ADDRESS(1),
 .UMR_CAPIM_TYPE(16'hA001),
 .UMR_DATA_BITWIDTH(UMR_DATA_BITWIDTH),
 .UMR_CAPIM_COMMENT_STRING("I_CAPIM_1"),
 .UMR_USE_LOCATION_ID(1))

capim_led_control (
 .umr_clk (umr_clkout),
 .umr_reset (),
 .wr(capim_data_wr),
 .dout(capim_data_do),
 .rd (capim_data_rd),
 .din(capim_data_di),
 .intr(1'b0),
 .inta (),
 .inttype(16'h0000)
) ;

.din(capim_data_di),
.intr(1'b0),
.inta (),
.inttype(16'h0000)
) ;
```

## Using UMRBus 3.0 with HAPS-100 Modules

See these topics for information about using UMRBus 3.0 with HAPS-100:

- [Using UMRBus 3.0 MCAPIM for HAPS-100 Modules](#), on page 455
- [Using UMRBus 3.0 Transactors](#), on page 457
- [Using JTAG Communication](#), on page 458

## Using UMRBus 3.0 MCAPIM for HAPS-100 Modules

Only UMRBus 3.0 MCAPIM (umr3\_mcapim\_ui) is supported for HAPS-100 modules. UMRBus 3.0 allows bi-directional communication with a higher bit width.

These are the guidelines:

- For mixed systems, partition UMRBus 3.0 MCAPIM to HAPS-100 and UMRBus 2.0 to HAPS-80.
- UMRBus 3.0 is backward compatible; replacing prior CAPIMs with the new ones. However new features, such as DMA, are not backward compatible.
- Do the following to convert HAPS-80 designs that use UMRBus 2.0 CAPIMs (`capim_ui` or `capim_wp`) to UMRBus 3.0 MCAPIM (`umr3_mcapim_ui`) for HAPS-100.
  - For a simple `capim_ui`, replace it with `umr3_mcapim_ui` and connect the UMRBus 3.0 ports. See the example that follows.
  - Remove or comment the parameter `//.UMR_USE_LOCATION_ID(1)`.
- You can downgrade the UMRBus 2.0 error to a warning by setting the environment variable `disable_umr2_capim_error` to TRUE.

## Example: UMRBus 2.0 to UMRBus 3.0

The following is an example of converting UMRBus 2.0 to UMRBus 3.0. The necessary changes are in bold.

### UMRBus 2.0

```
capim_ui #(.UMR_CAPIM_ADDRESS(UMR_CAPIM_BASE_ADDRESS) ,
 .UMR_CAPIM_TYPE(16'hD000) ,
 .UMR_DATA_BITWIDTH(UMR_DATA_BITWIDTH) ,
 .UMR_USE_LOCATION_ID(1) , .UMR_CAPIM_COMMENT_STRING("I_CAPIM_1"))

capim_led_control (
 .umr_clk (umr_clkout) ,
 .umr_reset (),
 .wr (capim_data_wr) ,
 .dout (capim_data_do) ,
 .rd (capim_data_rd) ,
 .din (capim_data_di) ,
 .intr (1'b0) ,
 .inta () ,
 .inttype (16'h0000)
);

 .din (capim_data_di) ,
 .intr (1'b0) ,
 .inta () ,
 .inttype (16'h0000)
);
```

```

capim_ui #(.UMR_CAPIM_ADDRESS(UMR_CAPIM_BASE_ADDRESS),
 .UMR_CAPIM_TYPE(16'hD000),
 .UMR_DATA_BITWIDTH(UMR_DATA_BITWIDTH),
 .UMR_USE_LOCATION_ID(1),
 .UMR_CAPIM_COMMENT_STRING("I_CAPIM_1")
)
capim_ua

 .umr_clk(umr_clk),
 .umr_reset(umr_reset),
 .wr(capim_data_wr),
 .dout(capim_data_do),
 .rd(capim_data_rd),
 .din(capim_data_di_ba),
 .intr(1'b0),
 .inta(),
 .inttype(16'h0000));

```

### UMRBus 3.0

```

umr3_mcapim_ui #(.UMR_CAPIM_ADDRESS(UMR_CAPIM_BASE_ADDRESS),
 .UMR_CAPIM_TYPE(16'hD000),
 .UMR_DATA_BITWIDTH(UMR_DATA_BITWIDTH),
 //.UMR_USE_LOCATION_ID(1),
 .UMR_CAPIM_COMMENT_STRING("I_CAPIM_1")
)

capim_ua (
 .umr_clk(umr_clk),
 .umr_reset(umr_reset),
 .wr(capi_dout_valid),
 .dout(capi_dout),
 .rd(),
 .din(capi_din),
 .intr(1'b0),
 .inta(),
 .inttype(16'h0000));

```

## Using UMRBus 3.0 Transactors

Follow these steps to convert existing HAPS-80 transactor designs to HAPS-100. If you use a new transactor design that is implemented for HAPS-100, XACTORS\_GEN automatically sets the right parameters and these steps are not required.

1. Add the define XACTORS\_UMR3 Verilog define as compile option.

2. Add following file to file list

```
$env(BUILD)/lib/synip/uc2/umrbus3/umrbus3.v
```

3. Comment out the umr\_clk\_reset instance in the top-level file.

```
// umr_clk_reset
// I_umr_clk_reset
//(
// .umr_clk_out(CCLOCK),
// .umr_reset_out(CRESET)
//);
```

## Using JTAG Communication

There is no dedicated JTAG connector on the HAPS-100 module. Instead the built-in UMRBus offers JTAG communication for your application. The following two step process uses the virtual cable (XVC) from Vivado Hardware Manager, which is a TCP/IP-based protocol that acts like a JTAG cable.

1. Start the JTAG server using Confpro commands on the host computer.

```
> cfg_jtag_start cfg0 fbl.ua [-port 65137]
> cfg_jtag_stop cfg0 fbl.ua
```

2. Connect to JTAG using the Vivado Hardware Manager.

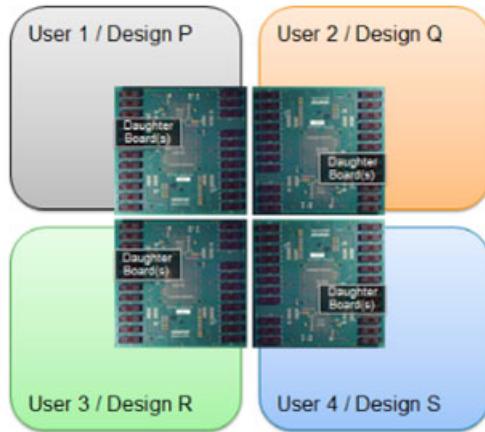
```
>connect_hw_server
>open_hw_target -xvc_url localhost: 65137
```

## Running Designs in Multi-Design Mode: HAPS-80 and HAPS-70

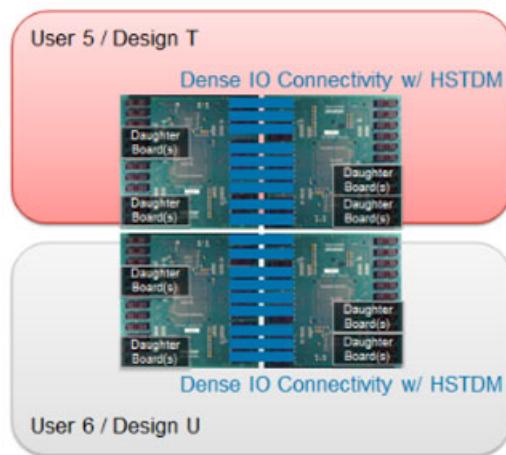
This procedure outlines how to set up designs for multi-design mode and how to run the designs for HAPS-70 and HAPS-80 systems.

1. Make sure you have the appropriate tools and hardware setup. You can have design setups as shown below:

4 single FPGA designs



Two 2-FPGA designs  
with HSTDMD



Multi-design mode is not limited to single FPGAs, but can be spread across multiple systems. For example, a 5-FPGA design can be implemented on FPGAs A, B, C, and D of one system and FPGA A of the second system, and another 3-FPGA design can be implemented on the remaining FPGAs (B, C, and D) of the second system.

Multi-design mode does not require cascaded systems. If they are not cascaded, remember that UMRBus GCLK0 will not be synchronized between systems through the CDE cable. If the CDE cable for UMRBus is connected, the systems are chained. The CDE CLK\_LEFT

and CLK\_RIGHT are configuration-dependent. See [Chaining Systems for MDM, on page 446](#) for more information.

If the systems are not chained, synchronize the user clocks independently, or use an ECDB to distribute them.

## 2. Plan the designs.

- Floorplan where the designs go. See [MDM Planning Guidelines for HAPS-70 and HAPS-80, on page 446](#) for more information.
- Plan the clocks and resets. See [Working with Design Resets for Multi-Design Mode, on page 449](#) and [Distributing Clocks for Multi-Design Mode, on page 450](#).

## 3. Instantiate CAPIM\_UI.

See [Instantiating CAPIM\\_UI, on page 452](#) for details about instantiating CAPIMs and the addresses you can use for them.

## 4. Set up configuration and queuing in Confpro.

Only one Confpro session is available per system. You can use a Confpro script to timeshare successfully.

- First, configure the system without any user designs. Configure the clocks and voltages.
- Configure the first user design. Use the umrbusscan command to report the addresses and check what is being used by the design.
- Configure the other designs one at a time. Subsequent umrbusscan commands report the addresses for all configured designs.
- Set up the designs to share time on the UMRBus, with sequential access.
- Use the Confpro -wait option to avoid lockout messages during system configuration. For example:

```
cfg_open haps80 emu:8 -wait
cfg_config_data cfg0 FB1_A myDesigns/top.bit
cfg_open haps80 emu:8 -wait
cfg_config_data cfg0 FB1_B myDesigns/FB1_uA/FB1_uA.bit
```

## 5. Run the designs sequentially.

If you are using the debug flow with MDM on HAPS-80 systems, be careful about using built-in memory for debug. Built-in memory is only available on FPGA A, so use it accordingly for MDM.

If you are using a transactor design, make sure to follow the guidelines in [Using Transactors in Multi-Design Mode, on page 451](#).

## MDM Planning Guidelines (HAPS-80 and HAPS-70)

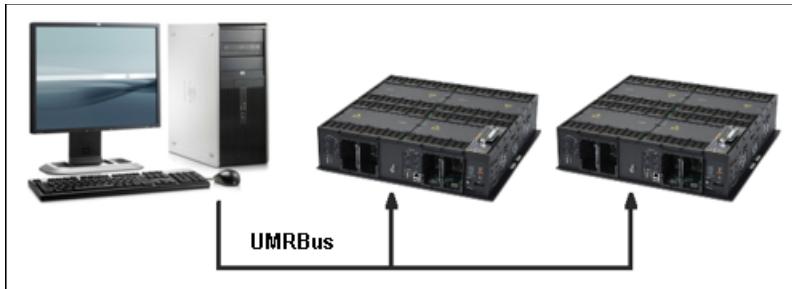
For multi-design mode, the designs need to be able to share and utilize the infrastructure of the HAPS systems. You must plan how design resets, design clocks, design UMRBus and CAPIMs will be shared; and how configuration and queuing is set up. See these topics for details:

- [Chaining Systems for MDM, on page 446](#)
- [Guidelines for Floorplanning MDM Designs, on page 447](#)
- [Working with Design Resets for Multi-Design Mode, on page 449](#)
- [Distributing Clocks for Multi-Design Mode, on page 450](#)
- [Using Transactors in Multi-Design Mode, on page 451](#)
- [Instantiating CAPIM\\_UI, on page 452](#)

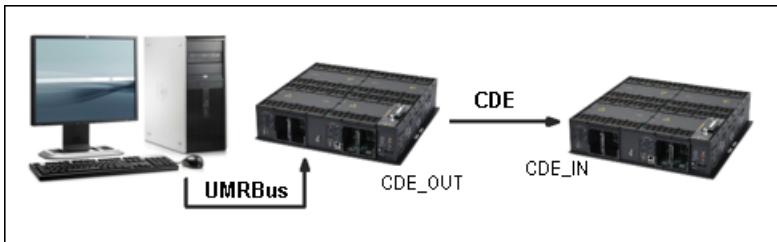
### Chaining Systems for MDM

The UMRBus chain defines the system for MDM. A single HAPS system is defined as a chained system on a single physical UMRBus.

- A chained system shares the same UMRBus resources (0 to 32). This means that two HAPS-80 S104 systems with separate UMRBus links are considered separate systems, even if they share the same host PC.



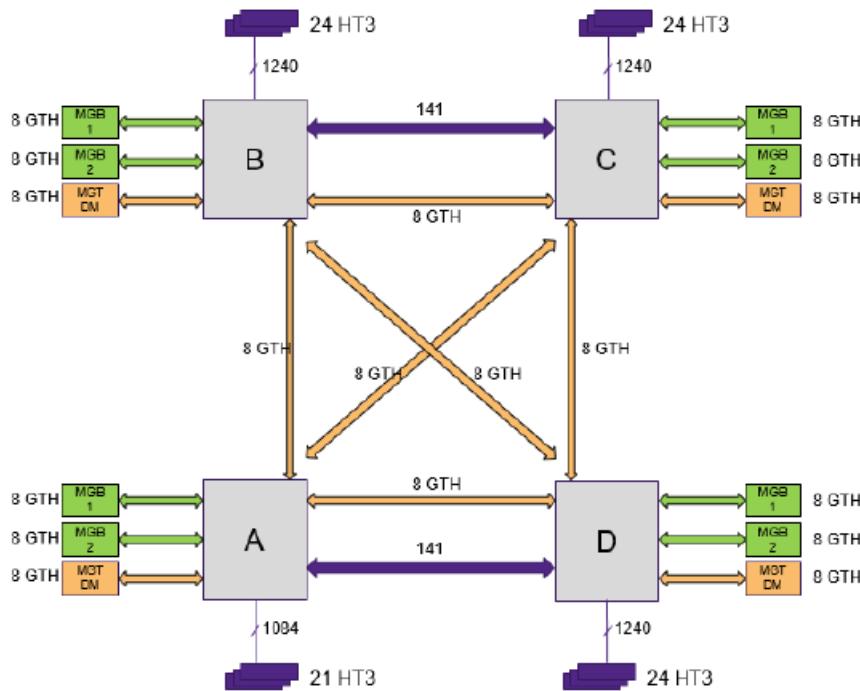
- A CDE cable connection in a chained system is considered a hard connection, because it is independent of the configuration setup. Two HAPS-80 S104 systems connected by CDE cable are treated as one HAPS-80 S208 system:



- GCLK0 is not synchronized through the CDE cable. If you need to synchronize it, use the clock connector definitions for the CDE to define soft connections that are configuration-dependent. You can also use a HAPS-ECDB to distribute the user clocks.
- One Confpro session is available per system, with each design on the system time-sharing the UMRBus, and accessing it sequentially.
- Initiate HSTDMD training from the Confpro GUI.
- You can configure a HAPS system without an FPGA binary, and add it in later. You can also configure one FPGA while you are using another.

## Guidelines for Floorplanning MDM Designs

- Each design can occupy any number of FPGAs. For example, a HAPS-80 S208 system supports multi-design combinations like the following: eight 1-FPGA designs; one 5-FPGA design and one 3-FPGA design; one 6-FPGA design and two 1-FPGA designs; and so on.
- Designs can spread across different HAPS systems. A single 5-FPGA design can be implemented on FPGAs A, B, C, and D of a HAPS-80 S104 and FPGA A of a second HAPS-80 S104 system.
- Multiple designs cannot share FPGAs or daughter boards.
- The designs can be the same design or unique.
- Multiple designs must share the same HAPS system infrastructure, like global clocks, design clocks and reset, UMRBus, and CAPIMs.



- For HAPS-80 and HAPS-100 designs, take advantage of the built-in interconnect architecture. The figure above shows the interconnects of the HAPS-80 S104. For two 2-FPGA designs on a HAPS-80 S104 for example, use FPGAs A and D for one design, and B and C for the other.
- Leverage the built-in features of the HAPS-80 architecture when planning for debug with MDM designs. Refer to the preceding figure.
  - To debug designs on individual FPGAs or pairs, use the built-in resources on FPGA A. See [Running Multi-FPGA Debug with HAPS-80 Built-in Memory, on page 712](#).
  - To run single-FPGA debug on all the FPGAs, add additional DDR memory boards as needed. See [Running Single-FPGA Debug on a HAPS-80 FPGA, on page 703](#).
  - If you do not want to use additional hardware, use synchronous GSV ([Using Global State Visibility \(GSV\), on page 753](#)). Use BRAM IICE to trigger the clock module. You can also use distributed BRAMs for multiple FPGAs.

## Working with Design Resets for Multi-Design Mode

For HAPS-70 systems, the global RESET<sub>n</sub> signal feeds all FPGAs in a one-CDE chain, or all four FPGAs of a HAPS-70 S48 system.

HAPS-80 systems are more flexible. Each FPGA has individual global RESET<sub>n</sub> signals. The signals can be globally or controlled individually for each FPGA through Confpro commands.

### 1. Set up the reset signals.

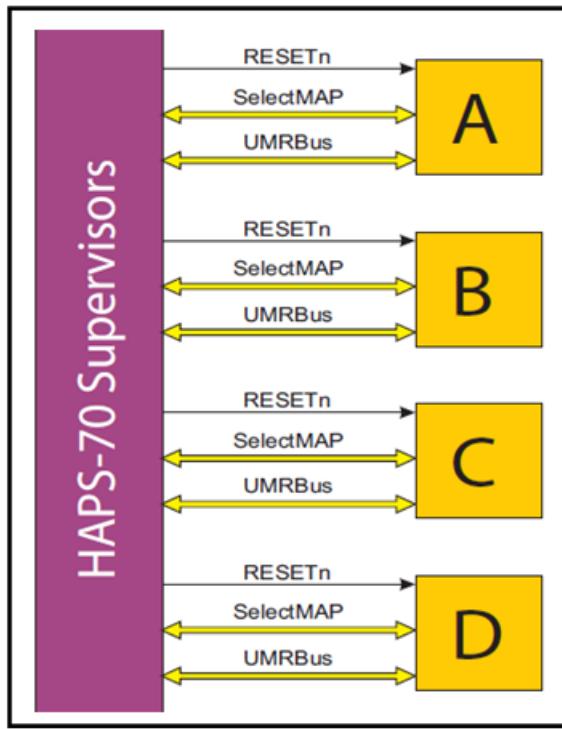
- For HAPS-80 single-system designs (four or fewer FPGAs), use the local reset generated from the System IP. Each FPGA has its own local reset. You can specify the resets globally or individually for each FPGA with the Confpro cfg\_reset\_set command.

```
cfg_reset_set cfg0 FB1 0
cfg_reset_set cfg0 FB1_B 1
```

- For HAPS-70 single-system designs (four or fewer FPGAs) use a GPIO daughter board or use Confpro to write to the CAPIM.
- For multi-FPGA designs with HSTDMD, use HAPS-controlled training ([Using HAPS-Controlled HSTDMD Training, on page 380](#)). For this case, use the GCLK to drive resets. Resets can only be driven from FPGAs A and B, so make sure that A and B are in separate pairs. The important thing to ensure is that when you use asynchronous resets, the two FPGAs are synchronized when they come out of reset.

### 2. Distribute the reset signals.

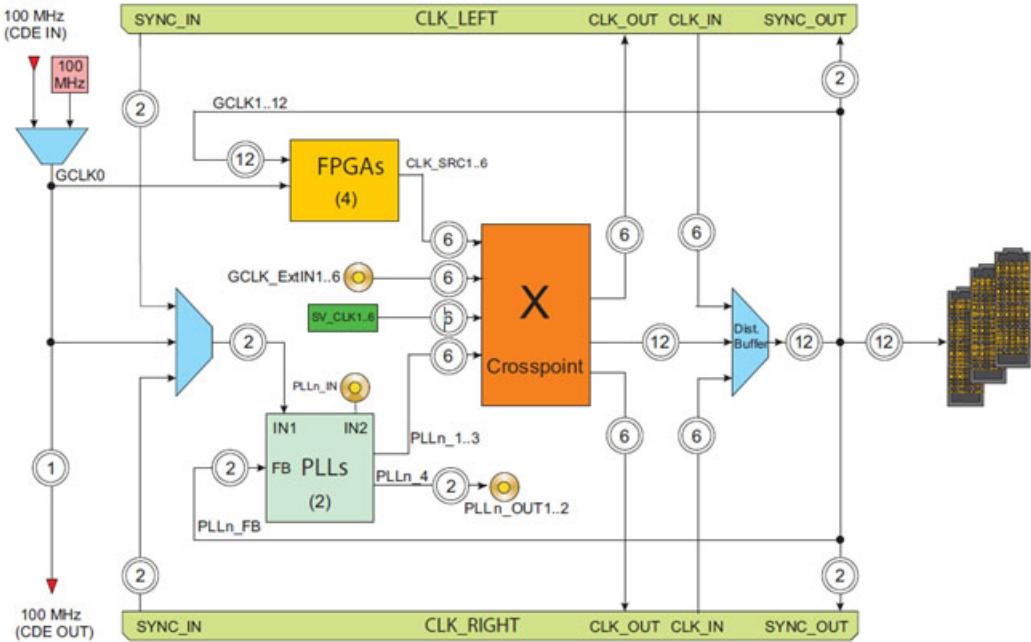
- For HAPS-80 systems, distribute the reset signal as described in [Assigning and Distributing HAPS-80 Reset, on page 278](#).
- For HAPS-70 systems, use a global clock instead of the global RESET<sub>n</sub> signal if you want independent controls for the designs. The RESET<sub>n</sub> signal feeds all the FPGAs in one CDE chain. On a HAPS-70 S48 system, the RESET<sub>n</sub> signal feeds all four FPGAs, as shown in the following figure.



## Distributing Clocks for Multi-Design Mode

Follow these guidelines when working with shared clocks in multi-use mode:

- Do not share clocks between designs. HAPS system clocks are common to all the FPGAs, as shown in the figure below. Plan your clock scheme accordingly.



- Avoid using clocks generated by the FPGAs because FPGA-generated clocks feed into the global clocks for the system. Do not use onboard inter-FPGA traces for MDM.

If you must use FPGA-generated clocks, review the implications of your setup before implementing it, and make sure you define it correctly, using FPGAs A or B as clock sources. See [Working with HAPS-80 Resets, on page 278](#) for details.

- If you have multiple copies of the same design configured on a HAPS system in multi-use mode, remember that the clocks are common to all the FPGAs in the system. Changes that affect the global clock setting affect the other FPGAs. So if you make a change, make sure to retain the global clock connections so that the same bit files can be reused. If you do not retain the original global clock connections, you must recompile the design to generate new bit files.

## Using Transactors in Multi-Design Mode

3. Regenerate transactors using the latest transactor package. Specify a global define to enable MDM:

```
'define UMR_USE_LOCATION_ID
```

4. Run the design and check the logs.
  - Check the log file after compilation to make sure the CAPIM\_UI is compiled (CG364). The file reports the location IDs of user CAPIMs.
  - After the pre-partition, partition, or system route stages, view the log file to check whether the address was implemented dynamically.

## Instantiating CAPIM\_UI

CAPIM\_UI supports HAPS-70 and HAPS-80 systems.

The CAPIM\_UI eliminates the need for multiple compile runs to address conflicts caused by multiple copies of the same bit file. CAPIM\_UI uses dynamic addressing instead of static addressing. The dynamic addresses are assigned by the supervisor at run time.

1. Instantiate CAPIM\_UI in your design.

Get the module definition from the /umr\_capim.v file, located in *installDir/lib/vlog*. The CAPIM address must include the FPGA location ID and non-zero MSB bit address. See [CAPIM UI Example, on page 455](#).

CAPIM\_UI is the default for HAPS-80 systems, but not for HAPS-70.

The number of CAPIMs that can be shared in multi-user mode depends on the hardware setup:

| System Size                    | CAPIMs/FPGA                        | Valid CAPIM Addresses |
|--------------------------------|------------------------------------|-----------------------|
| 4 or fewer<br>chained systems  | 7<br>(Including transactor CAPIMs) | 1, 2, 3, 4, 5, 6, 7   |
| More than 4<br>chained systems | 3<br>(Including transactor CAPIMs) | 1, 2, 3               |

Note that the available CAPIM addresses are different from non-MDM mode. In non-MDM mode, 32 CAPIMs can be used for the user design, addresses 1-31.

2. Check that the UMR\_USE\_LOCATION\_ID parameter in the CAPIM definition is set to 1 to use dynamic addressing.

```
capim_ui #(
 .UMR_CAPIM_ADDRESS(1),
 .UMR_CAPIM_TYPE(16'hA001),
 .UMR_DATA_BITWIDTH(UMR_DATA_BITWIDTH),
 .UMR_CAPIM_COMMENT_STRING("I_CAPIM_1"),
 .UMR_USE_LOCATION_ID(1))
```

If it is set to 0, the CAPIM address is defined by UMR\_CAPIM\_ADDRESS.

- Dynamic addressing avoids address conflicts. Dynamic addressing is the default for HAPS-80 systems but not for HAPS-70 systems. When you configure multiple designs with UMRBus, pay special attention to the CAPIM IDs, and make sure that you do not unintentionally run into the other designs and cause address conflicts. Instead of using static addresses, the supervisor sets the CAPIM address dynamically at runtime according to the FPGA where the CAPIM\_UI is partitioned, thus eliminating address conflicts.

The FPGA name is important to identify the FPGA when configuring with Confpro, and for debug. In the table, FPGA names follow the convention FBX\_Y, where X is the system number in the CDE chain, and Y is the FPGA name (A-D).

| FPGAs     | Dynamic FPGA Location ID (XXX) |
|-----------|--------------------------------|
| FB[1-4]_A | 000                            |
| FB[1-4]_B | 001                            |
| FB[1-4]_C | 010                            |
| FB[1-4]_D | 011                            |
| FB[5-8]_A | 100                            |
| FB[5-8]_B | 101                            |
| FB[5-8]_C | 110                            |
| FB[5-8]_D | 111                            |

With dynamic addressing, the UMR\_CAPIM address is handled with fpga\_location\_id (based on the FPGA), rather than as a static address.

{fpga\_location\_id[2:0], UMR\_CAPIM\_ADDRESS [1:0] }

- For HAPS-70 systems to use dynamic addressing based on location, you must also set a global attribute in the FDC file. It is not required for HAPS-80 systems.

```
define_global_attribute syn_umr_use_capim_wp 1
```

- For HAPS-70 systems with the older CAPIMs, avoid address conflicts by manually assigning different base addresses for the FPGAs in each idc file. Otherwise you might encounter No instrumented bit file found errors.

```
fpga_A.idc file: device capimbaseaddr 10
fpga_B.idc file: device capimbaseaddr 12
```

3. Reduce the number of HAPS systems in chain with the define UMR\_CHAIN\_LIMIT4 parameter.

The default chain limit is 8, where as user CAPIMs are shared between pairs of HAPS systems (first and fifth; second and sixth, third and seventh; fourth and eighth) with a maximum of 3 user CAPIMs per FPGA.

If you set UMR\_CHAIN\_LIMIT4, the number of HAPS systems in the chain is 4, so the number of user CAPIMs per FPGA increases. There are now a maximum of 7 user CAPIMs per FPGA.

If the number of user CAPIMs is a problem, consider using separate systems. For example, use two 4-FPGA systems instead of a chained 8-FPGA system. Handle GCLK0, which is synchronized through the CDE.

4. After running pre-partition, check the CAPIM addresses in the report log to make sure that no static addresses are assigned.

The address must show CAPIM\_WP:

| CAPIM   | Bus | FPGA | Width | Addr     | Type     | Comment        |
|---------|-----|------|-------|----------|----------|----------------|
| I_capim | -   | -    | 8     | CAPIM_WP | CAPIM_WP | I_gpio_master  |
| I_capim | -   | -    | 8     | CAPIM_WP | CAPIM_WP | I_axi4_master2 |
| I_capim | -   | -    | 8     | CAPIM_WP | CAPIM_WP | I_axi4_master1 |

## CAPIM\_UI Example

```
capim_ui #(
 .UMR_CAPIM_ADDRESS(1),
 .UMR_CAPIM_TYPE(16'hA001),
 .UMR_DATA_BITWIDTH(UMR_DATA_BITWIDTH),
 .UMR_CAPIM_COMMENT_STRING("I_CAPIM_1"),
 .UMR_USE_LOCATION_ID(1))

capim_led_control (
 .umr_clk (umr_clkout),
 .umr_reset (),
 .wr(capim_data_wr),
 .dout(capim_data_do),
 .rd (capim_data_rd),
 .din(capim_data_di),
 .intr(1'b0),
 .inta (),
 .inttype(16'h0000)
) ;

.din(capim_data_di),
.intr(1'b0),
.inta (),
.inttype(16'h0000)
) ;
```

## CHAPTER 5

# Running the Implementation Flow

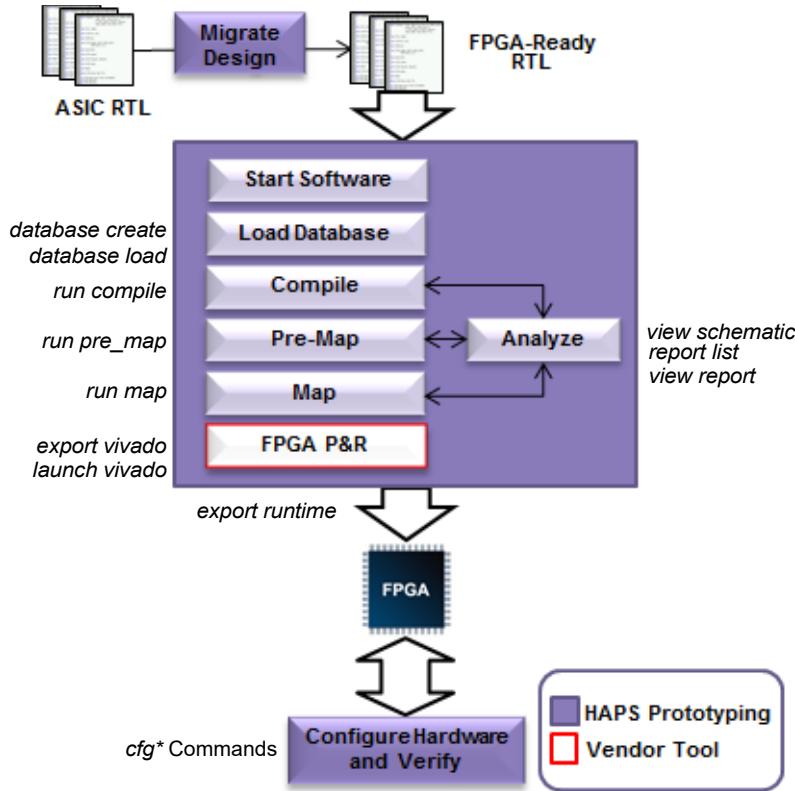
---

This chapter describes procedures for a basic flow through synthesis, placement, and routing, without debugging. For information about a debug flow, see *Instrumenting and Debugging Designs*, on page 633.

- [The Basic Implementation Flow](#), on page 472
- [Running Pre-Map](#), on page 474
- [Mapping to Hardware](#), on page 481
- [Mapping the Design](#), on page 490
- [Synthesizing Based on Design Intent](#), on page 493
- [Using Distributed Processing](#), on page 497
- [Analyzing Results](#), on page 512
- [Checking Timing Results](#), on page 528
- [Handling Errors and Warnings](#), on page 550
- [Running Simulation](#), on page 557
- [Running Place and Route](#), on page 576
- [Analyzing Congestion Issues](#), on page 589
- [Running in Batch Mode](#), on page 603
- [Working with Tcl Scripts and Commands](#), on page 609
- [Preparing to Run on the Hardware](#), on page 614
- [Working with Mixed-System Designs](#), on page 601

# The Basic Implementation Flow

The following figure shows the steps in the basic design flow to implement an FPGA, along with the main commands used at each step. For details about the commands, refer to the *Command Reference Manual*.



Some steps have been described earlier and are not repeated in this chapter. The following table points to the information for each step in the flow.

|                |                                                                                                     |
|----------------|-----------------------------------------------------------------------------------------------------|
| Migrate Design | <a href="#">Migrating Designs for Synthesis, on page 819</a>                                        |
| Load Database  | <a href="#">Working with a Design Database, on page 58</a>                                          |
| Compile        | <a href="#">Adding Design Files, on page 96</a><br><a href="#">Compiling the Design, on page 77</a> |
| Premap         | <a href="#">Running Pre-Map, on page 474</a>                                                        |

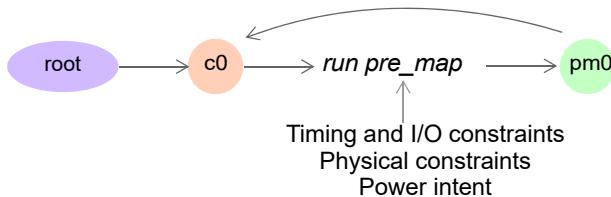
---

|                               |                                                                           |
|-------------------------------|---------------------------------------------------------------------------|
| Map                           | <a href="#">Mapping the Design</a> , on page 490                          |
| FPGA Place and Route          | <a href="#">Running Place and Route without Exploration</a> , on page 580 |
| Analyze Results               | <a href="#">Analyzing Results</a> , on page 512                           |
| Export to Bit File            | <a href="#">Exporting Files for Runtime</a> , on page 614                 |
| Configure Hardware and Verify | <a href="#">Ensuring Hardware Bring-up</a> , on page 621                  |

---

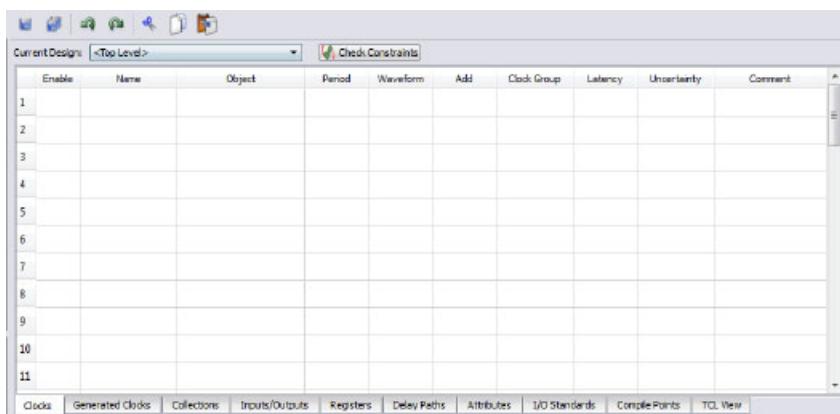
## Running Pre-Map

After you have compiled the design, you are ready to run the pre-map phase of synthesis. At this stage, you specify constraints that optimize timing, set signal-to-pin assignments, specify power intent, or implement post-compile substitutions in the gate-level netlist.



The following procedure shows you how to implement this phase as an individual step, but you can also run it as part of a complete synthesis process based on design goals, by following the steps described in [Synthesizing Based on Design Intent, on page 493](#). For tools other than ProtoCompiler DX, this step could also be included as part of a generated Tcl script for synthesizing partitioned FPGAs in a multi\_FPGA design.

1. Start with a compiled design.
2. Set timing constraints in an existing fdc file or create a new one.
  - Use edit fdc -mode gui to open the GUI where you can specify the constraints. This opens the SCOPE® spreadsheet, where you can edit constraints and add attributes. Alternatively, use a text editor to create or edit an fdc file.



- You must define the global clock with the `create_clock` constraint. For example: `create_clock {gclk0} -period {0}`.
  - Include the `fdc` file as input to the pre-mapper with the `-fdc` argument to the `run pre_map` command (step 6).
3. Set synthesis attributes for I/O planning in the `fdc` file.
    - If you are using the GUI to define the attributes, go to the Attributes tab in the SCOPE spreadsheet.
    - Specify I/O locations and I/O buffer configurations in an `fdc` file with the `syn_loc` and `syn_pad_type` constraints. You must specify these constraints to place and route successfully. See [Mapping to Hardware, on page 481](#) for more information about pin location constraints, especially for `gclk0`.
    - Include the `fdc` file as input to the pre-mapper with the `-fdc` argument.
  4. To manage the assignment of signals and pins to HAPS connectors, do the following:
    - To assign RTL design signals to HAPS-70 connectors in single-FPGA designs, specify HAPS pin-mapping constraints in the `fdc` file. Use `define_haps_io` constraints to map Xilinx device pins (AX23 for example) to HAPS-70 board connector locations such as `J1A[0]`.

The following syntax shows the basic `fdc` syntax for HAPS pin mapping, where `portname` is the signal name on a top-level netlist port, and `hapsConnectorName` is the corresponding HAPS connector name.

```
define_haps_io {p:portname} -haps_io {hapsConnectorName}
```

See [Mapping to HAPS Pins, on page 481](#) for more information, and [define\\_haps\\_io, on page 298](#) in the *Command Reference Manual* for the complete syntax.

When you define constraints with `define_haps_io`, at the mapping stage the design uses the assigned HAPS pin names instead of the device pin names. The tool also generates a report file called `haps_io_report.txt` under the premap database, and you can view the RTL and HAPS views of the assignments. You can export the `haps_io_report.txt` file with `export reports -all`.

If you make assignments to invalid HAPS I/O names, you get an error message. The `define_hap_io` constraints become location constraints for Vivado place and route.

For tools other than ProtoCompiler DX, `define_haps_io` applies to the mapper and is intended for use in single-FPGA designs where there is no partitioning. If it is a partitioned design and you do not want the partitioner to assign traces to the I/O ports, you must specifically set `port_attribute` in the pcf file so that the partitioner ignores `define_haps_io`:

`port_attribute -ignore [portBusNameList]`

- Specify the physical location of the signal defined with `define_haps_io` using the `define_haps_fpga_location` constraint in the fdc file.
  - If your design uses GCLK0, set constraints as defined in [Working with GCLK0, on page 486](#).
  - To automatically infer differential I/Os in HAPS-70 systems, assign the pair to locations that correspond to the I/O differential pair of the target technology part and package, and specify the correct I/O standard for that part and package. Use `define_haps_io` or `syn_loc`, and `syn_pad_type` to do this. For details, see [Inferring Differential Pairs, on page 483](#).
5. Optionally, specify power requirements using the UPF format.
- Define power domains and specify isolation and retention requirements in a UPF file. See [Including UPF Specifications, on page 827](#) for detailed information.
  - Add the UPF file to the design with the `load_upf` command, or specify the file with the `-upf` argument when you run the pre-mapper.
6. Specify other synthesis options.
- Specify options, like automatic conversion of gated clocks, using the `option set` command in a Tcl file. See [Pre-Map Options, on page 479](#) for the options you can set at this design stage.
  - For single-FPGA designs, make sure to specify the technology and speed grade with the `option set technology` and `option set speed_grade` commands, respectively.

For partitioned FPGAs, you must define these options before partitioning. Define the technology when you create the database (`database create`) and the speed grade in the TSS file (`board_system_create`). Specify one speed grade per HAPS system; you can use different speed grades in the same setup. Do not change these inherited settings at the individual FPGA level.

- Save the Tcl options file and then source it. When you source the file, the options specified in it apply to this database state and the ones below it in the database tree.

The first command launches a Tcl editor to create an options file called MyPrepOptions.tcl, and the second sources the file:

```
edit file MyPrepOptions.tcl
source file MyPrepOptions.tcl
```

7. Start with a compiled design and run pre-mapping with the `run pre_map` command, or click Pre-Map in the left panel to run it from the GUI.
  - If you have FDC constraints, specify them with the `-fdc` argument to the command. For example: `run pre_map -fdc myConstraints.fdc`.
  - If you have UPF constraints, put them in a `upf` file and specify it with the `-upf` argument to the command.

When you run this command, the tool generates a new database node, called `pm0` by default. The tool generates a log file and a schematic, which you can view in a schematic window to analyze the results of the run.

8. Analyze the results of the reports and view the schematic.
  - To view the I/O report with the HAPS pin mapping assignments, select it from the Report View in the GUI from the list of reports generated after the pre-mapping stage. The RTL View section of the report lists the RTL ports and their corresponding HAPS connectors. The HAPS View section lists all legal HAPS I/Os, with the RTL bit ports, connectors, direction, and type.

Report View

Report: synregtest\_db (root)

- [-] c0 (compile)
  - [+] Compile Log
  - [+] multi\_srs\_gen
- [-] pm0 (pre\_map)
  - [+] HAPS IO Report**
  - [+] Pre\_Map Log
- [-] m0 (map)
  - [+] Map Log

RTL View for Design h70\_basic  
Target Platform: HAPS-70

| RTL Port | HAPS Connector |
|----------|----------------|
| pin_C12  | A1             |
| pin_C11  | A1             |
| pin_N12  | A1             |
| pin_M12  | A1             |
| pin_clk  | A23            |
| pin_in1  | A23            |
| pin_out1 | A23            |

---

HAPS View

| HAPS Connector | HAPS Pin    | RTL |
|----------------|-------------|-----|
| MGB1           | AM1_TXN [3] |     |
| MGB1           | AM1_RXP [3] |     |
| MGB1           | AM1_TXP [3] |     |
| MGB1           | AM1_RXN [2] |     |
| MGB1           | AM1_TXN [2] |     |
| MGB1           | AM1_RXP [2] |     |
| MGB1           | AM1_TXP [2] |     |
| MGB1           | AM1_RXP [2] |     |
| MGB1           | AM1_TXP [2] |     |

To export the file, first use the report list command to view the list of reports and get the name of the file. Then go to the premapped database state and use export report to export the HAPS I/O report.

- See [Analyzing Results, on page 512](#) for more information about checking the log file and other results of the run.
9. Rerun premapping as needed until the design is properly constrained and intermediate logic substitutions and optimizations are properly applied.

You can then proceed to mapping your design ([Mapping the Design, on page 490](#)).

## Pre-Map Options

You can specify options for the pre-map stage of synthesis as arguments to the run pre\_map command or as arguments to the option set command. You can collect the latter in a Tcl file and then source the Tcl file at the command line.

| Option                                                                                | Description                                                                                                                                                                                                                                                                                                                                                                                                                      |
|---------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Command Arguments (run pre-map Command)                                               |                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| -fdc <i>fileName</i>                                                                  | Loads the specified FDC file, and uses the defined constraints in the pre-map phase.                                                                                                                                                                                                                                                                                                                                             |
| -upf <i>fileName</i>                                                                  | Loads the specified UPF file and uses the power constraints specified in the pre-map phase. See <a href="#">Including UPF Specifications , on page 827</a> for information about using this file.                                                                                                                                                                                                                                |
| Option Settings (option set). Click the Options Editor icon for a comprehensive list. |                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| automatic_compile_point 0 1                                                           | Determines whether the tool automatically divides up the design into compile points, without creating additional hierarchy. Compile points are RTL partitions of the design that are treated as independent blocks by the synthesis engine.                                                                                                                                                                                      |
| fix_gated_and_generated_clocks 0 1                                                    | Determines whether the gating is separated from the clock inputs in gated clocks, and whether generated clock logic is replaced with logic that uses an initial clock with an enable. The default, 0, does not specify these conversions. For details about gated clock conversion, see <a href="#">Converting Gated Clocks , on page 862</a> .                                                                                  |
| force_async_genclk_conv 1 0                                                           | Determines whether generated clocks are converted, and whether datapath latches that are driven by generated clocks are converted to a flip-flop and multiplexer when the latch is set or reset by asynchronous signals. By default, the latches with asynchronous control signals are not converted. See <a href="#">Defining Clocks for Gated Clock Conversion , on page 872</a> for information about its use and an example. |
| max_parallel_jobs_number                                                              | Determines the maximum number of processing jobs to run in parallel. Each license allows up to four parallel jobs. Use this in conjunction with automatic_compile_point to reduce runtime.                                                                                                                                                                                                                                       |
| Constraints (fdc File)                                                                |                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| define_haps_io                                                                        | Maps device pins to HAPS connector locations.                                                                                                                                                                                                                                                                                                                                                                                    |
| define_haps_fpga_location                                                             | Specifies pin location on HAPS system for signals defined with define_haps_io.                                                                                                                                                                                                                                                                                                                                                   |

| Option                | Description                                                                                                                                                                                                                                                                     |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Attributes (fdc File) |                                                                                                                                                                                                                                                                                 |
| syn_loc               | Assigns signals to I/O pin locations for implementation during place-and-route. Use the pin table and trace names sections of the hardware <i>Reference Manual</i> to get the pin designations.<br>See <a href="#">Example of I/O Constraints in an fdc File</a> , on page 487. |
| syn_pad_type          | Defines I/O buffer configuration.<br>See <a href="#">Example of I/O Constraints in an fdc File</a> , on page 487.                                                                                                                                                               |

# Mapping to Hardware

This section describes some hardware I/O planning considerations to take into account when describing the hardware for pre-map and map operations.

See the following for more information:

- [Mapping to HAPS Pins, on page 481](#)
- [Inferring Differential Pairs, on page 483](#)
- [Working with GCLK0, on page 486](#)
- [Setting up GPIOLINK Connections \(HAPS-70 and HAPS-DX7\), on page 488](#)
- [Working with HAPS Systems, on page 269](#)

## Mapping to HAPS Pins

The prototyping tool includes mechanisms to define the mapping to HAPS pins.

### Defining HAPS Pins in Multi-FPGA Designs

For multi-FPGA designs, defining HAPS pins is a two-part process:

1. Define the hardware connections in the tss file. For example:

```
board_system_create -interconnect -manual DDR3_SODIMM2R_HT3
-name DDR3_MEM -connector {fb1.A9 fb1.A10 fb1.A11}
```

2. Assign the pins with constraints in the pcf file. For example:

```
assign_port {ddr3_addr[15:0]} -trace {DDR3_MEM.DDR3_A[15:0]}\nassign_port {ddr3_ba[2:0]} -trace {DDR3_MEM.DDR3_BA[2:0]}\nassign_port {ddr3_dq[63:0]} -trace {DDR3_MEM.DDR3_DQ[63:0]}
```

Do not use the `define_haps_io` constraint to define HAPS pins in multi-FPGA designs; it is intended for use with single-FPGA designs only. For partitioned designs where you do not want the partitioner to assign traces to the I/O ports, you must specifically set this `port_attribute` command in the pcf file:

```
port_attribute -ignore [portBusNameList]
```

## Defining HAPS Pins in Single-FPGA Designs with define\_haps\_io

For single-FPGA designs, follow these steps to define HAPS pins:

1. Check for pin names.

- Run pre-map (run pre\_map) and check the haps\_io\_report, which is generated for each FPGA after the command runs. The report gives you an idea of which HAPS pins to use.
- For specific details about the I/O interface, consult these sources:

|                                                 |                                                   |
|-------------------------------------------------|---------------------------------------------------|
| Pin, bank, and associated connector information | <i>HAPS Reference Manual</i> for your HAPS system |
|-------------------------------------------------|---------------------------------------------------|

|                          |                                                                                           |
|--------------------------|-------------------------------------------------------------------------------------------|
| Pinout and I/O resources | Xilinx: Packaging and Pinout Product Specification<br>FPGA SelectIO Resources: User Guide |
|--------------------------|-------------------------------------------------------------------------------------------|

If you make assignments to invalid HAPS I/O names, you get an error message.

2. Set define\_haps\_io constraints in the fdc file to define the HAPS pins. For example:

```
define_haps_io {p:portname} -haps_io {$hapsConnectorName}
```

When you define constraints with define\_haps\_io, at the mapping stage the design uses the assigned HAPS pin names instead of the device pin names. The tool also generates a report file called haps\_io\_report\_txt under the premap database, and you can view the RTL and HAPS views of the assignments. You can export the haps\_io\_report\_txt file with export reports -all. The define\_hap\_io constraints become location constraints for Vivado place and route.

You can define the I/O connections as Tcl variables to make it more flexible. This example shows Tcl variables used in the define\_haps\_io constraints, which makes it easy to make modifications in one place:

```
set DDR3_JX1 A3
set DDR3_JX2 A4
set DDR3_DM "B\[7:0\]"
```

```
define_haps_io {p:DDR3_DM_TRY1[7:0]} -haps_io ${DDR3_JX1}_${DDR3_DM}
define_haps_io {p:DDR3_DM_TRY2[7:0]} -haps_io ${DDR3_JX2}_${DDR3_DM}
```

For more about using `define_haps_io` constraints, see these links:

---

|                                     |                                                                                     |
|-------------------------------------|-------------------------------------------------------------------------------------|
| Syntax                              | <a href="#">define_haps_io</a> , on page 298 in the <i>Command Reference Manual</i> |
| Differential pairs                  | <a href="#">Specifying Differential Pairs with define_haps_io</a> , on page 483     |
| Differential clocks                 | <a href="#">Specifying Differential Clocks</a> , on page 486                        |
| GCLK location constraints           | <a href="#">Working with GCLK0</a> , on page 486                                    |
| Running pre-map with the constraint | <a href="#">Running Pre-Map</a> , on page 474                                       |

---

## Inferring Differential Pairs

For HAPS-70 systems, you can automatically infer ibufds for differential input signals, ibufgds for differential clocks, iobufds for differential bidirectional I/O, and obufds for differential output. To automatically infer differential pairs, you must specify two things: the I/O pair location on the target part package, and the I/O standard for that package.

There are two ways to specify the pin location. The recommended way is to use `define_haps_io`, but you can alternatively use the `syn_loc` attribute. Use the `syn_pad_type` attribute to specify the I/O standard, except for differential clocks, which do not require the I/O standard to be specified.

### Specifying Differential Pairs with `define_haps_io`

This is the recommended method for HAPS-70 systems. To use `define_haps_io` to specify constraints for input, output, or bidirectional differential pairs, follow these steps:

1. Look up the pin tables in the documentation for the HAPS hardware you are using to find the HAPS differential pin locations that correspond to the Xilinx pin locations you want to use. The Xilinx file shows the pin information for the part and package. For example:

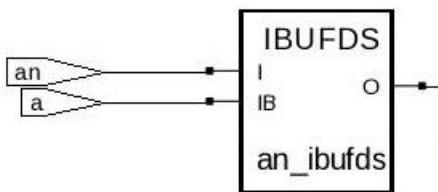
| Pin  | Pin Name              |
|------|-----------------------|
| AU30 | IO_L12P_T1_MRCC_36 HP |
| AU31 | IO_L12N_T1_MRCC_36 HP |

However, when you specify the I/O constraints for the differential pair with `define_haps_io`, you must use the HAPS locations that correspond to the Xilinx pins, as shown in the example below.

2. In the fdc file, specify the HAPS differential pair pin location with the define\_haps\_io constraint.
3. Specify the I/O standard for the package with the syn\_pad\_type attribute.

```
define_haps_io {p:a} -haps_io {A7_B[0]} -comment {"Xilinx AU30"}
define_haps_io {p:an} -haps_io {A7_B[1]} -comment {"Xilinx AU31"}
define_io_standard {p:a} syn_pad_type {LVDS_18}
 -delay_type {input}
define_io_standard {p:an} syn_pad_type {LVDS_18}
 -delay_type {input}
```

For the example above, the tool automatically converts the define\_haps\_io constraints to Xilinx PACKAGE\_PIN constraints in the xdc file, and infers an ibufds (shown below). The xdc file is generated after pre\_map. Consult the Xilinx documentation for details about the I/O banks and the I/O standard to specify.



- Specify other I/O standard attributes with define\_io\_constraint in the fdc file, as required by the standard. Additional information, like syn\_io\_dci and syn\_io\_termination, are required for certain standards, to direct the tool to interpret them correctly. For example, you must set them for Vivado HSTL\_II\_T\_DCI; if you do not specify the additional attributes as shown below, the tool applies the HSTL\_II standard instead of what you intended.

```
syn_pad_type {HSTL_Class_II},
define_io_constraint syn_io_dci {DCI}
define_io_constraint syn_io_termination {Thevenin}
```

4. Check the haps\_io\_report.txt file after pre-map.

The run pre\_map command generates a haps\_io\_report.txt file. The top of the report shows the RTL port name and the HAPS connector:

| RTL View for Design h70_basic |                |
|-------------------------------|----------------|
| Target Platform : HAPS-70     |                |
| <hr/>                         |                |
| RTL Port                      | HAPS Connector |
| pin_clk                       | A23            |
| pin_inl                       | A23            |
| pin_out1                      | A23            |
| pin_gclk                      | GCLK           |

The HAPS View section lists all the HAPS connectors, HAPs pins, RTL pin names, I/O standards, corresponding Xilinx pins, SLR, and comments added to the `define_haps_io` constraint:

| HAPS View      |            |              |         |          |              |            |          |     |         |
|----------------|------------|--------------|---------|----------|--------------|------------|----------|-----|---------|
| HAPS Connector | HAPS Pin   | RTL Pin Name | IO Std  | Function | TDM Type     | Xilinx Pin | Bank     | SLR | Comment |
| MGB1           | AH1_RXN[3] |              |         | MGT      |              | AG3        | MGT: 115 | 1   |         |
| MGB1           | AH1_RXP[3] |              |         | MGT      |              | AH6        | MGT: 115 | 1   |         |
| ....           |            |              |         |          |              |            |          |     |         |
| A23            | A23_D[18]  | pin_inl      |         |          |              | BB36       | 11       | 0   |         |
| A23            | A23_D[9]   | pin_out1     |         |          |              | BD34       | 11       | 0   |         |
| ....           |            |              |         |          |              |            |          |     |         |
| GCLK           | GCLKIN[0]  |              |         |          | GCLKB_100MHz | AE41       | 14       | 1   |         |
| GCLK           | GCLKP[0]   | pin_gclk     | LVDS_18 |          | GCLKB_100MHz | AD41       | 14       | 1   |         |
| ....           |            |              |         |          |              |            |          |     |         |

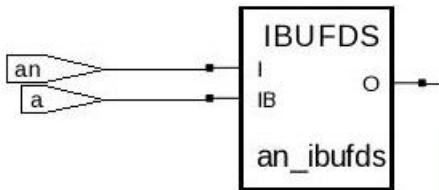
## Specifying Differential Pairs with Attributes

To use attributes to specify constraints for input, output, or bidirectional differential pairs, do the following:

1. Specify the Xilinx differential pair pin location, not the HAPS location, with `syn_loc`.
2. Specify the I/O standard for the package with the `syn_pad_type` attribute.

```
define_attribute {p:a} {syn_loc} {AU30}
define_attribute {p:an} {syn_loc} {AU31}
define_io_standard {p:a} syn_pad_type {LVDS_18}
 -delay_type {input}
define_io_standard {p:an} syn_pad_type {LVDS_18} -delay_type
{input}
```

The tool infers an ibufds:

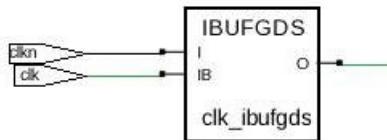


## Specifying Differential Clocks

To specify constraints for differential clocks, use the `define_haps_io` command:

```
define_haps_io {p:pin_gclk} -haps_io {GCLKP[0]}
```

You do not need to specify the I/O standard. The tool automatically infers the differential pair, GCLKP[0] and GCLKN[0] in this example, and inserts the ibufgds.



It also creates xdc constraints like the following:

```
set_property PACKAGE_PIN AD41 [get_ports {pin_gclk}]
set_property PACKAGE_PIN AE41 [get_ports {pin_gclk_0}]
set_property IOSTANDARD LVDS [get_ports {pin_gclk}]
```

## Working with GCLK0

There are some considerations to keep in mind if the design uses GCLK0:

1. Define GCLK0 location constraints.

For single-FPGA designs that require global clocks, the recommendation is to set pin location constraints on gclk0. This is because some automatically inserted IPs use the gclk0 signal. If your design also uses gclk0, this could cause errors during place and route. For example, if you use gclk0 for manually instantiated single-ended buffers or MMCMs or PLLs with different configurations, this could conflict with the requirements for the

inserted IP. Inserting location constraints resolves the conflict by informing the tool of this use.

Use one of these two ways to set location constraints:

- Add the `define_haps_io` constraint to the FDC file to set a location constraint for `gclk0`. This is the recommended method.

```
define_haps_io {p:gclk0} -haps_io {GCLKP[0]}
define_haps_io {p:gclk0_n} -haps_io {GCLKN[0]}
define_haps_io {p:reset_n} -haps_io {RESET_n}
define_haps_io {p:gpiolink_out} -haps_io {GPIOLINK_OUT}
define_haps_io {p:gpiolink_in} -haps_io {GPIOLINK_IN}
```

See [Example of I/O Constraints in an fdc File, on page 487](#).

- Set `syn_loc` and `syn_pad_type` attributes to define the location. The following example directs place-and-route to assign the input port `gclk0`, a differential input clock signal, to pins AV33 and AV34 with a LVDS I/O standard. The remaining signals are assigned to HapsTrak 3 locations and the LVCMOS 1.8V I/O standard:

```
define_attribute {p:gclk0} syn_loc {AV33}
define_attribute {p:gclk0_n} syn_loc {AV34}
define_io_standard {p:gclk0} syn_pad_type {LVDS}
define_io_standard {p:gclk0_n} syn_pad_type {LVDS}
```

You do not need to set locations for partitioned FPGAs because the software automatically does this from the top level.

## 2. Set the clock frequency to 100 MHz.

If your design uses `gclk0`, set a 100 MHz constraint on this clock. You must do this because this clock is fixed to 100 MHz in the hardware.

### Example of I/O Constraints in an fdc File

```
Global 100 MHz clock input
create_clock {gclk0} -period {10}

I/O site locations
define_attribute {p:gclk0} syn_loc {AV33}
define_attribute {p:gclk0_n} syn_loc {AV34}
define_attribute {p:reset_n} syn_loc {AV32}
define_attribute {p:gpiolink_out} syn_loc {BD34}
define_attribute {p:gpiolink_in} syn_loc {BD35}
```

```

I/O buffer configuration
define_io_standard {p:gclk0} syn_pad_type {LVDS}
define_io_standard {p:gclk0_n} syn_pad_type {LVDS}
define_io_standard {p:reset_n} syn_pad_type {LVC MOS18}
define_io_standard {p:gpiolink_out} syn_pad_type {LVDCI_DV2_18}
define_io_standard {p:gpiolink_in} syn_pad_type {LVC MOS18}

HAPS I/O constraints
define_haps_io {p: b_ten_in[3:0]} -haps_io
 {A10_A[2],A10_A[3],A10_A[4],A10_A[5]}
define_haps_io {p:b_dut_in1} -haps_io {A11_B[9]}

```

## Setting up GPIOLINK Connections (HAPS-70 and HAPS-DX7)

The HAPS-DX7 and HAPS-70 systems do not include physical pins for user GPIO, user LED, and MGB GPIO. Instead, a serialization GPIOLINK IP block that is instantiated in the user design through the physical pins `GPIOLINK_IN` and `GPIOLINK_OUT` handles these signals. For details about GPIOLINK, refer to the hardware documentation for the system.

Note the following details:

1. To use GPIOLINK, instantiate it in your design.

The IP block is included with the prototyping software installation, in the `install/lib/synip/gpio/hapsdx7/verilog` directory or the `install/lib/synip/gpio/verilog` directory.

The GPIO IP includes a new output FPGA location ID, a 3-bit location ID based on the HAPS FPGA to which the bit file is configured. The `reset_n` input is no longer inside the IP, but is automatically reset using `umr_reset`. You can connect this input to any signal.

2. Use big-endian syntax [1:10] to define the GPIOLINK ports, as shown in the instantiation example below:

```
wire [1:10] haps_MGB1_GPIO_in;
```

3. To simplify the setting of constraints, the recommendation is to use `LVDCI_DV2_18` for all UMRBus and GPIOLINK signals.

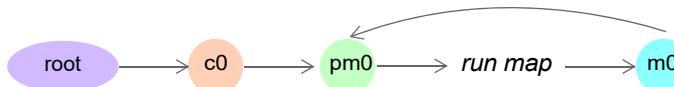
You can use either `LVDCI_DV2_18` or `LVC MOS` for the inputs, but the outputs must be `LVDCI_DV2_18`. See [Example of I/O Constraints in an fdc File, on page 487](#) for an example.

4. If you use an MGB card, you must use the I/O port names from the connected riser card in the instantiation, not the I/O ports from the PCIe card.

For example, if you use GPIOLINK for PCIe sideband signals, the GPIOLINK instantiation must use the I/O port names for the MGB riser card connector that the PCIe MGB card is plugged into, not the I/O port names from the PCIe card itself. To get the names for your particular hardware setup, consult the pin tables in the appropriate hardware document, which in this case is the *PCIE-4-KIT\_MGB Reference Manual*.

# Mapping the Design

Mapping is the last synthesis phase of FPGA synthesis, where the logic is mapped to technology-specific resources. At this stage, you can set options to convert and optimize gated and generated clocks, and for packing.



The following procedure shows you how to run this phase individually, but you can run the entire synthesis process based on design goals, by following the steps described in [Synthesizing Based on Design Intent, on page 493](#).

1. Start with a design that has gone through the pre-map stage ([Running Pre-Map, on page 474](#)).

If needed, use the appropriate database commands to close the current database and load an appropriate database node. See [Loading and Navigating Database States, on page 62](#) for more information.

2. Specify synthesis mapping options in a Tcl file. Save the Tcl options file and then source it.

The first command launches a Tcl editor to create an options file called MyMapOptions.tcl, where you can enter option set commands. The second command sources the file:

```
edit file MyMapOptions.tcl
source file MyMapOptions.tcl
```

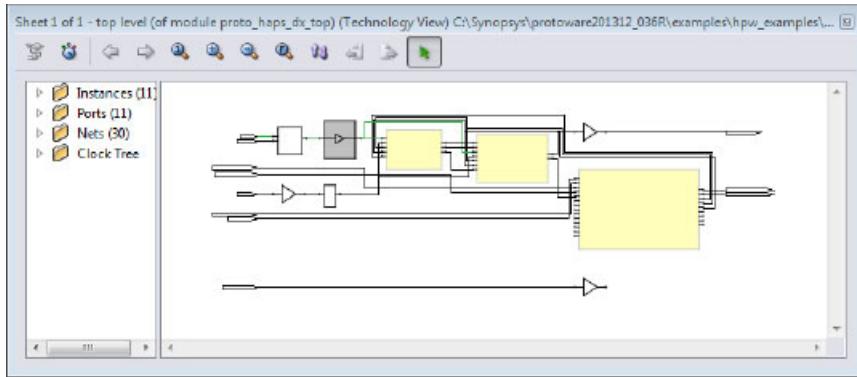
See [Mapping Options, on page 491](#) for the options you can set at this design stage.

3. Enter the run map command: run map

When you run this command, the tool generates a new database node, called m0 by default. The tool synthesizes the design and maps the logic to technology-specific resources. In addition to the log file, the tool also generates an srm file, which you can view in a schematic window to analyze the results of the run. The schematic shows design logic as well as support functions like UMRBus and initialization blocks for the hardware system.

4. Analyze the results.

See [Analyzing Results, on page 512](#) for more information about checking the results of the run.



In particular, check these results: clock conversion, timing performance, cell and I/O usage, and hierarchical area usage.

5. Rerun mapping as needed to refine gated clock conversion and optimize the packing of logic into resources.

When you are satisfied with system performance and resource utilization, you can perform one of these operations:

Place and route exploration [Running Place and Route, on page 576](#)

Place and route [Running Place and Route without Exploration, on page 580](#)

Simulation [Running Simulation, on page 557](#)

Equivalence checking [Verifying the Design, on page 937](#)

## Mapping Options

| Option                                                                                | Description                                                                                                                                  |
|---------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| Option Settings (option set). Click the Options Editor icon for a comprehensive list. |                                                                                                                                              |
| automatic_compile_point 1 0                                                           | Determines whether the tool automatically creates compile points in the design.                                                              |
| retiming 1 0                                                                          | Determines whether registers can be moved into combinational logic to improve performance. The default is 1, which allows this optimization. |

| Option                                       | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| continue_on_error                            | Specifies whether the mapping operation continues instead of stopping when it encounters errors in a compile point. When enabled (1, the default), the tool continues mapping the rest of the design. See <a href="#">Using Continue on Error , on page 820</a> for details.                                                                                                                                                                                                                       |
| synthesis_strategy advanced fast routability | <ul style="list-style-type: none"> <li>• advanced – Performs additional optimizations to provide an output netlist with better QoR, but may require longer runtimes compared to other modes.</li> <li>• fast – Runs the mapper in a fast mode that is optimized for prototyping.</li> <li>• routability – Performs placement-aware congestion reduction techniques to produce a routable netlist. There is a trade-off in timing QoR to improve synthesis runtime. This is the default.</li> </ul> |
| no_sequential_opt 1 0                        | Determines whether sequential optimizations are performed. The default is 0, where optimizations are performed.                                                                                                                                                                                                                                                                                                                                                                                    |
| enable_prepacking 1 0                        | Determines whether the synthesis netlist uses advanced LUT combining, so that they can be packed more efficiently during place-and-route. This option is enabled (1) by default.                                                                                                                                                                                                                                                                                                                   |
| verification_mode 1 0                        | Enables or disables formal verification with the Formality tool. It is disabled by default. When enabled, the tool generates an SVF file for equivalence checking.                                                                                                                                                                                                                                                                                                                                 |
| fix_gated_and_generated_clocks 1 0           | Enables or disables gated and generated clock optimizations. This option is enabled by default.                                                                                                                                                                                                                                                                                                                                                                                                    |
| force_async_genclk_conv 1 0                  | Determines whether generated clocks are converted, and whether datapath latches that are driven by generated clocks are converted to a flip-flop and multiplexer when the latch is set or reset by asynchronous signals. By default, the latches with asynchronous control signals are not converted. See <a href="#">Defining Clocks for Gated Clock Conversion , on page 872</a> for information about its use and an example.                                                                   |

# Synthesizing Based on Design Intent

The prototyping tool includes an intuitive GUI where you can execute a complete synthesis run (compile, pre-map, map) with options that are optimized for the design goal you specify.

See the following procedures for details:

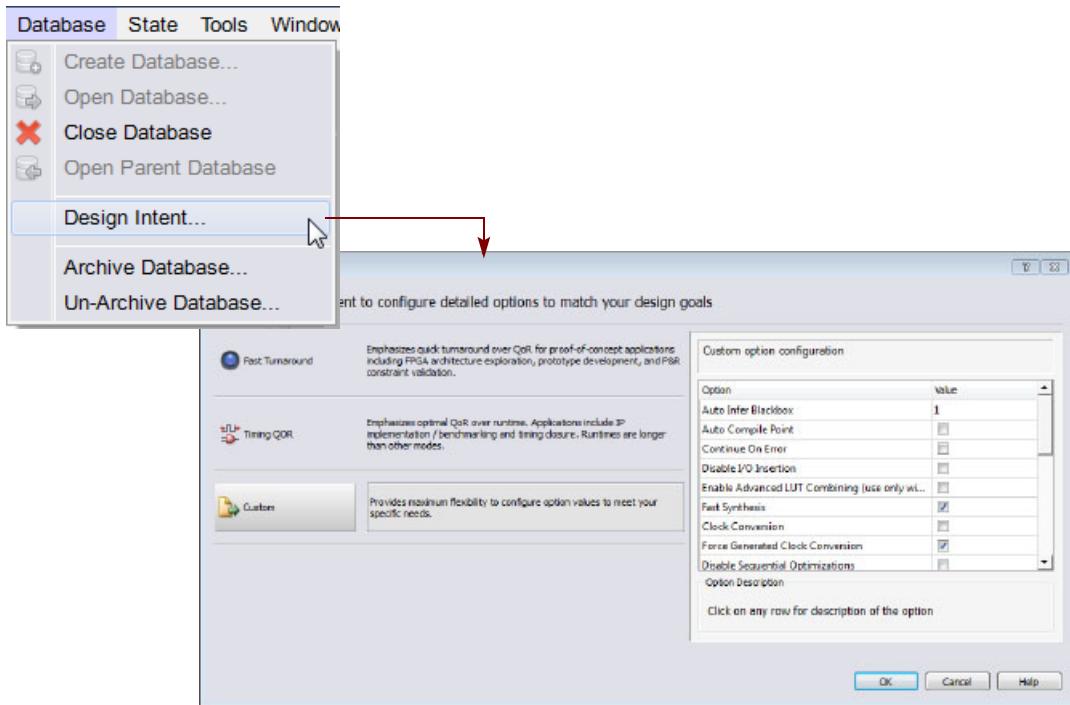
- [Using Design Intent to Synthesize a Quick Prototype](#), on page 493
- [Synthesizing for Optimal QoR Using Design Intent](#), on page 494

## Using Design Intent to Synthesize a Quick Prototype

The tool offers a shortcut for a quick, first-pass synthesis run. You can do this to generate an initial prototype, or to estimate how to allocate resources and divide up the design. The following procedure shows you how to signify design intent from either the command line and the GUI.

1. To specify design intent without using the GUI, load a database and then do the following:
  - Specify the `design_intent fast_turnaround.tcl` command. This command opens the named file located in `/lib/design_intent/protocompiler`; the file contains the default set of options for quick synthesis. You can synthesize using these default options.
  - To use a custom set of options, specify the `design_intent -custom` command. This command opens the `fast_turnaround.tcl` file, and you can use this as a template and customize the options. Save the options file, and then synthesize using the customized set of options.
2. Load a database and select Design Intent from the Database popup menu.

The Design Intent window opens.



3. To use the default settings for a quick prototype, click **Fast Turnaround** and then click **OK**.

The tool automatically sets the best options to minimize runtime and ensure a quick turnaround and synthesizes the design with these options. The table on the right of the dialog box shows the settings used.

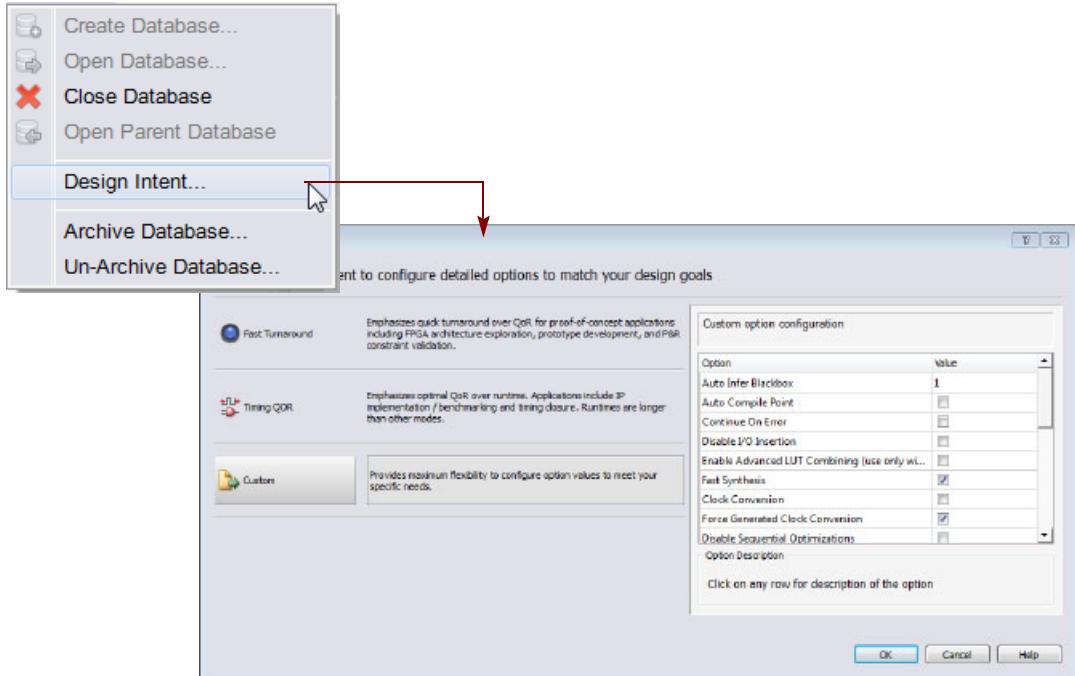
This mode uses the TTFP (Time To First Prototype) compiler, which is a thin compiler that is faster than the default compiler. The compiler generates a schematic where you can check the results.

4. To customize the options, click **Custom**, select the option settings you want, and then click **OK**.

## Synthesizing for Optimal QoR Using Design Intent

The synthesis tool offers a shortcut to setting options to achieve the best QoR for the prototype. You can set design intent from the GUI or from the command line, as described in the following procedure:

1. To specify design intent without using the GUI, load a database and do the following:
  - Specify the `design_intent timing_qor.tcl` command. This command opens the named file located in `/lib/design_intent/protocompiler`; the file contains the default set of options for best results. You can synthesize using these default options.
  - To use a custom set of options, specify the `design_intent -custom timing_qor.tcl` command. This command opens the default `timing_qor.tcl` file, which you can use as a template to customize the options. Save the options file, and then synthesize using the customized set of options.
2. To use the GUI, load a database and select Design Intent from the Database popup menu.



3. To use the default settings, click Timing QoR and click OK.

When you synthesize, and place and route the design, the tool automatically sets the best options to for an at-speed prototype and synthesizes

the design with these options. The table on the right of the dialog box shows the settings used.

4. To customize the options, click Custom, select the option settings you want, and then click OK.

# Using Distributed Processing

Distributed or parallel processing is a technique used to reduce runtime. With this technique you divide and conquer, splitting up processes so that they can run in parallel instead of serially, thus saving time.

The number of processes run in parallel is determined by the number of available licenses and available jobs, as well as the `-max_parallel_jobs` option.

See the following topics for more information:

- [Distributed Processing Schemes](#), on page 497
- [Distributed Processing Operations](#), on page 497
- [Setting Options to Run Distributed Processing](#), on page 498
- [Using CDPL for Distributed Processing](#), on page 500
- [Running Distributed Compile](#), on page 503
- [Running Distributed Synthesis](#), on page 506
- [Running Place-and-Route Exploration](#), on page 576
- [Synthesizing Individual FPGAs Using a Script](#), on page 426

## Distributed Processing Schemes

There are two schemes for multiprocessing:

- With the CDPL (Common Distributed Processing Library) scheme:  
CDPL distributes processes across multiple machines
- Without CDPL:  
Processes are distributed among the processors on a single machine

## Distributed Processing Operations

The following table lists processes that can be processed in parallel in distributed mode.

| Process                                                | Usage                                                                                                                                                                                                           |
|--------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Distributed compile                                    | Limitations: multiple configurations, hierarchical defparam, signed parameters                                                                                                                                  |
| Distributed synthesis                                  | Limitations: constraint checker, continue on error, and compile points are ignored.                                                                                                                             |
| Compile point processing                               | Compile points are ignored in distributed synthesis mode, but can otherwise be processed in parallel. See <a href="#">Chapter 5, Working with Compile Points</a> in the <a href="#">Compiler Mapper Guide</a> . |
| Exploratory place and route runs                       | See <a href="#">Running Place-and-Route Exploration</a> , on page 576 for details                                                                                                                               |
| Quick area estimation                                  | Enable with the <code>run pre_partition -area_est 1</code> command.                                                                                                                                             |
| Multiple tool invocations to process partitioned FPGAs | Use the <code>launch protocompiler -max_parallel_jobs</code> command.                                                                                                                                           |

## Setting Options to Run Distributed Processing

There are various factors that affect a distributed processing run, like the number of tool licenses, the maximum number of jobs to multiprocess (four jobs in parallel per license), the distribution scheme, and which processes that can be run in parallel.

You can specify distributed processing when you start a run with the `launch` command, or set it as an option. The following procedure describes how to do this.

1. Set the maximum number of jobs to run.

There are two ways to set the number of jobs, depending on the operation you want to multiprocess:

- (HAPS-80) Use the `option set` command to set another value (the default is four per license) for distributed processing in the compile and map phases. This value is used instead of the default to process compile points, compiler runs, or synthesis operations:

```
option set max_parallel_jobs num_of_jobs
```

See [Running Distributed Synthesis](#), on page 506 for details.

Set the `-max_parallel_jobs` value to a number greater than 1, the default. There are four jobs available per tool license. If you have multiple tool licenses, correspondingly more jobs become available.

- (HAPS-100) Use the `option set` command to set another value (the default is four per license) for distributed processing in the compile and map phases a single FPGA flow. This value is used instead of the default to process compile points, compiler runs, or synthesis operations in a single FPGA flow:

```
option set max_parallel_jobs num_of_jobs
```

See [Running Distributed Synthesis, on page 506](#) for details.

For HAPS-80, The number of licenses works in tandem with the number of jobs to determine the actual number of jobs. If you set the maximum number of jobs to 10 and you only have two tool licenses, the total possible number of jobs that can be run in parallel is eight. Similarly, if you have three tool licenses but only five available jobs, the total possible number of jobs is five.

Note that some processes might consume additional resources. Check the log file for current license usage, to get an idea of what is available.

For HAPS-100, four jobs are available for each tool license. If you have multiple tool licenses, correspondingly more jobs will be available.

- Specify the `-max_parallel_jobs` argument to the `launch protocompiler` command to run multiple invocations of the tool and process the partitioned FPGAs in parallel. If you do not specify this command argument, the tool uses the default value of four per license or the value specified in the `max_parallel_jobs` option, if that was specified.

In the following HAPS-80 example, the first script runs with the default number of jobs or the number set previously with the `max_parallel_jobs` option. For the second script, 8 jobs run in parallel, because the `max_parallel_jobs` command argument is specified.

```
launch protocompiler -script fpga1.tcl -max_parallel_jobs 8 -script fpga2.tcl
```

In the following HAPS-100 example, the first script runs with the default number of jobs, which is 4. For the second script, 8 jobs run in parallel, because the `max_parallel_jobs` command argument is specified.

```
launch protocompiler -script fpga1.tcl -max_parallel_jobs 8 -script fpga2.tcl
```

2. If you want to use CDPL to distribute processing jobs over multiple Linux machines, set up the CDPL machines and enable the CDPL option (option set cdp1 1).

See [Using CDPL for Distributed Processing, on page 500](#) for details. If you do not use CDPL, the tool distributes processing among multiple processors on the same machine.

3. To run distributed synthesis or distributed compile, set additional options as described in [Running Distributed Compile, on page 503](#) and [Running Distributed Synthesis, on page 506](#).
4. To distribute area estimation runs, specify the -area\_est 1 argument with the run pre\_partition command.
5. To launch multiple tool invocations to process partitioned FPGAs in parallel, use the launch protocompiler -max\_parallel\_jobs command.

See [Synthesizing Individual FPGAs Using a Script, on page 426](#) for details.

6. To multiprocess compile points, enable the compile point option and disable distributed synthesis.

If distributed synthesis is enabled, compile points are not run in parallel during the map phase.

7. To parallel-process PAR explorer runs, use the option set max\_parallel\_par\_explorer 1 command..

See [Running Place-and-Route Exploration, on page 576](#) for details.

## Using CDPL for Distributed Processing

You can specify that the tool use the Synopsys CDPL (Common Distributed Processing Library) scheme for distributed processing or parallel processing. CDPL is a mechanism for deploying multiple processors to run processes like compiling and mapping in parallel on different machines. It also allows you to launch third-party applications, like place-and-route for example, if you have the requisite license set up.

CDPL consists of a master-worker distributed processing framework that distributes jobs across multiple machines. It only runs on Linux platforms. For Windows, or Linux machines without CDPL, the tool supports parallel processing on a single machine using multiprocessing.

For comprehensive information about CDPL, refer to the *Common DP Library User Guide*, a PDF version of which is included in the doc directory of the tool installation. The following procedure summarizes what you need to do to use CDPL in the prototyping tool. If you do not use CDPL, by default the tool distributes the processing jobs among the available processors in the current machine.

1. Specify the Linux machines to use for CDPL.

CDPL only runs on Linux platforms.

Create a hosts file that lists the machines for distributed processing, using the following syntax to specify them:

```
flag|hostname | slots | tmpDir | protocol | command
```

Four parallel processing jobs are allowed per license. If you have multiple licenses, you can specify more distributed processing jobs. See [Host File Examples for Distributed Processing, on page 502](#) for an explanation of the syntax and examples.

2. Set the CDPL\_FPGA\_HOST environment variable to point to the CDPL executable and libraries:

```
csh setenv CDPL_FPGA_HOST hostFilePath
```

```
bash, ksh export CDPL_FPGA_HOST=hostFilePath
```

Make sure to specify an absolute path. Using a relative path causes a failure.

3. In the prototyping tool, do the following:

- Set the number of jobs to run in parallel with the -max\_parallel\_jobs option. You can set this as an option with the option set command to distribute compile and map processes, or specify it as an argument to a the launch protocompiler command to process FPGA partitions in parallel.

Each tool license lets you run up to four jobs in parallel. The actual number of jobs that run in parallel depends on the number of available licenses and available jobs as well as the `-max_parallel_jobs` value.

- Enable CDPL with the `cdpl` option:

```
option set cdpl 1
```

You can now leverage CDPL for the parallel processing of various operations. See [Running Distributed Compile, on page 503](#) and [Running Distributed Synthesis, on page 506](#) for information about running various distributed synthesis processes.

When CDPL is specified, the following command distributes the running of multiple scripts across the specified CDPL machines to implement the partitioned FPGAs:

```
launch protocompiler -max_parallel_jobs 2 -script path/slp1.tcl -script path/slp2.tcl
-max_parallel_jobs 10-script path/slp3.tcl
```

## Host File Examples for Distributed Processing

Host files specify machines for distributed processing using this syntax:

```
flag|hostname | slots | tmpDir | protocol | command
```

|                 |                                                                                                                                                                                            |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>flag</b>     | Specifies whether a machine can be used. Valid values: <ul style="list-style-type: none"><li>• 0 signifies the host cannot be used</li><li>• 1 signifies a usable worker machine</li></ul> |
| <b>hostname</b> | Name of a valid worker machine. This field is empty for the SGE and LSF protocols.                                                                                                         |
| <b>slots</b>    | The maximum number of worker slots available on the host for RSH and SSH protocols, and in the farm for LSF and SGE. -1 indicates unlimited worker slots for LSF and SGE protocols.        |

---

|          |                                                                                                                                                                                                                                          |
|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| tmpDir   | Any directory, to be used for local data storage as required.                                                                                                                                                                            |
| protocol | Protocol used to connect to the worker machine. Valid values are RSH, SSH, SGE, and LSF, for Remote Shell, Secure Shell, Sun Grid Engine and Load Sharing Facility, respectively.                                                        |
| command  | Corresponding connection command for the protocol, which is used to connect to the worker machine. If protocol is defined as RSH, rsh is the corresponding connection command. For LSF and SGE, you can also specify command parameters. |

---

## RSH Host File Example

The following host file specifies a five-worker, two-node configuration for a distributed processing run. The tool launches up to two workers on the sbgemtin01-25 machine, and up to three workers on the sbgemtin01-22 machine.

```
1|sbgemtin01-25|2|/myDir/cdpl/tmp|RSH|rsh
1|sbgemtin01-22|3|/myDir/cdpl/tmp|RSH|rsh
```

## SGE Host File Example

You must properly configure the end-user environment, and it must be capable of starting SGE jobs. This is an example of an SGE host file entry:

```
1| |32|/tmp|SGE|qsub -P bnormal -l os_bit=64,mem_free=64G,h_vmem=64G
```

## LSF Host File Example

You must properly configure the end-user environment, and it must be capable of starting LSF jobs. This is an example of an LSF host file entry:

```
1| |32|/tmp|LSF|bsub -q bnormal -R os_bit==64 -R "rusage[mem=32000]"
```

## Running Distributed Compile

You can run distributed compile on single-FPGA or multi-FPGA designs. In multi-FPGA designs, use distributed compile for both the top-level RTL compile process as well as the FPGA-level compile.

Distributed compilation can be run as an initial run or incrementally. It is recommended that you run distributed compile on all designs to improve runtime.

1. If you are using CDPL for distributed processing, make sure the setup is in place and you have enabled the CDPL option.

See [Using CDPL for Distributed Processing, on page 500](#) for details. If you do not specify CDPL, the tool uses available processors on the current machine to distribute the jobs. You can try running distributed synthesis without CDPL first and then try it using CDPL.

2. Specify the number of compiler jobs to distribute and run in parallel.

The default value is 4, but you increase the value with this command, according to the computing resources you have available.

```
option set max_parallel_jobs num_of_jobs
```

the number of computing resources available.

3. Set the distributed compile option:

- Use this command to specify distributed compilation:

```
option set distributed_compile 1
```

This option only runs distributed compile; for information on distributed mapping, see [Running Distributed Synthesis, on page 506](#).

- Specify the distributed compile option last, after the options described in the previous steps.

4. Compile the design as usual with the run compile command.

The tool splits the design into smaller parts (distribution points or nodes) and compiles the nodes in parallel. It creates separate post-compile database files for each group. The log file reports the separate processes that are being distributed with messages like this:

```
Compiling work_cm3_dts_verilog as a separate process
Compiling work_cm3_sched_verilog as a separate process
Compiling work_cm3_srb_verilog as a separate process
...
...
```

The top-level log file contains a **Distributed Compiler Report**, which shows the status of each group. This is an example of an entry in that report:

| DP Name              | Status  | Start time | End Time   | Total Real Time | Log File                               |
|----------------------|---------|------------|------------|-----------------|----------------------------------------|
| work_cm3_dts.verilog | Success | 0h:00m:01s | 0h:00m:03s | 0h:00m:02s      | synwork//<br>distcomp<br>distcompl.log |

5. Check for errors and black boxes.

- If you get an error for a node during distributed compilation, you can bypass the node by setting the CDC `syn_distcomp_node` directive on the failing module:

```
define_directive {v:libName.moduleName} {syn_distcomp_node} {0}
```

The directive forces the compiler to create new groups for the next run, and to avoid using the failing node as a distribution node.

Re-compile the design after setting the directive.

- If black boxes are created after distributed compile, it means that the tool encountered linking issues. Typically, this happens with certain constructs (see [Distributed Compile Limitations, on page 506](#)). You can either run in non-distributed mode or bypass the failing node, as described above.

6. Check the names after running distributed compile.

Distributed compilation might create different names than non-distributed compile, so if you need to set constraints, make sure that the names match the names generated after distributed compile.

7. To run distributed compile incrementally, do the following:

- Make any changes required.
- Rerun distributed compile. The tool only recompiles and generates new post-compile databases for groups where changes were made and their parents. This results in runtime savings on iterative passes, depending on the number of nodes affected. If the change is to a top-level parameter or a common file, like package, the recompile will affect many more nodes, and there might not be much of a runtime advantage.

## Distributed Compile Limitations

There are some limitations to keep in mind when running distributing compilation.

### UC and Distributed Compile

The UC flow only supports distributed compiler mode and runs in this mode regardless of the distributed compile setting. If the design cannot be divided into parallel processes for the distributed compiler, it runs in monolithic mode and reports the following message:

@N: This design does not have sufficient nodes/jobs for distribution, running in monolithic mode.

### XMRs

Do not define XMRs in a cdc file; you must specify them in the RTL. If you do not specify them in the RTL, the tool issues a warning, and runs in non-distributed mode.

### Constructs

Some constructs cause linking issues:

- Some complex parameters, such as unions
- Sign extensions with signed parameter override

The tool creates black boxes when it encounters linking issues. In such cases, either run compilation in non-distributed mode, or bypass failing nodes by setting `syn_distcomp_node` to 0.

## Running Distributed Synthesis

To speed up runtime, run synthesis operations in a distributed fashion. When you follow the procedure below, the tool runs mapping processes in parallel. You can run distributed synthesis on single FPGAs, so that run `pre_map` and run `map` operations run in parallel.

1. If you are using CDPL for distributed processing, make sure the setup is in place and you have enabled the CDPL option.

See [Using CDPL for Distributed Processing, on page 500](#) for details. If you do not specify CDPL, the tool uses available processors on the current machine to distribute the jobs. You can try running distributed synthesis without CDPL first and then try it using CDPL.

2. Check that the following option values have been set:

- Use option set `max_parallel_jobs num_of_jobs` to specify the number of jobs to process in parallel. The default is 4. Do this for both distributed compile and distributed synthesis.
- For distributed synthesis, make sure the following options are also set:

| Option Setting                                      | Description                                                                                                 |
|-----------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <code>option set max_parallel_jobs numOfJobs</code> | Number of jobs to process in parallel. The default is 4, but it must be set to a higher value like 8 or 12. |
| <code>option set automatic_compile_point 0</code>   | Controls the automatic creation of RTL partitions (compile points).                                         |

Make sure to disable the automatic compile point options; otherwise distributed synthesis will not run.

3. Set the distributed synthesis options and synthesize.

- Set this option in a single-FPGA design to specify distributed synthesis:

```
option set distributed_synthesis 1
```

This option runs the mapping operations from the pre-map and map stages in parallel.

- Make sure that the distributed processing option is specified last, after all other option settings described in the previous steps.
- Run the mapping phases (`run pre_map`, `run map`) as usual on the single FPGA. The tool runs synthesis processes like constraint application, clock conversion, and mapping in distributed mode. It reports the separate processes being run in the log file.

See [Clock Conversion and Distributed Synthesis, on page 510](#) for information about how clock conversion is handled.

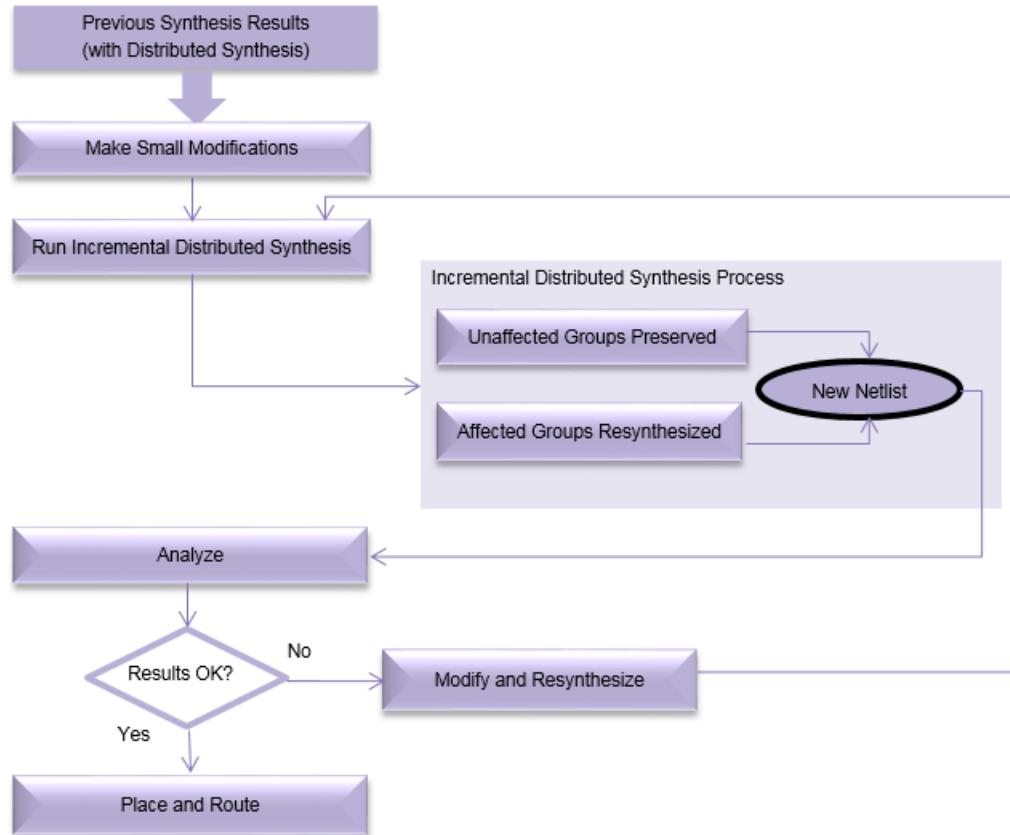
Distributed processing affects some synthesis features. See [\*Distributed Synthesis Limitations\*, on page 510](#) for details.

- To distribute the synthesis of partitioned FPGAs in a multi-FPGA design, launch and run scripts in parallel, as described in [\*Synthesizing Individual FPGAs Using a Script\*, on page 426](#).
- 4. To run distributed synthesis incrementally, do the following:
  - After the initial distributed synthesis run, make the changes you need to fix errors or improve your design.
  - Set the incremental distributed synthesis option.

`option set -incremental_synth 1`

- Rerun the map stages to resynthesize the design using distributed processing (run pre\_map, run map).

When the design is resynthesized, only those synthesis groups whose source code logic has been modified are resynthesized, using distributed processing. The mapped files generated from the previous run are used for the remaining, unchanged groups. If you modify existing constraints or options, the entire design is resynthesized.



- Check the log file for changes.

The following figure shows the results of an incremental distributed synthesis run. After the first run, logic changes were made to the design. This figure shows that incremental synthesis only resynthesized the top-level `eight_bit_uc` and `prgmcntr` modules.

| BN:MF797 :                    | Total number of groups created: 9         |            |                |                  |
|-------------------------------|-------------------------------------------|------------|----------------|------------------|
| BN:MF796 :                    | Maximum number of parallel jobs set to 2  |            |                |                  |
| =====Summary for Mapping===== |                                           |            |                |                  |
| Hierarchical Instance Name    | Mapping Status                            | Total Time | Total CPU Time | Peak Memory used |
| sc_alu                        | Unchanged Reused previously mapped design | 0h:00m:03s | 0h:00m:00s     | 165 MB           |
| dmux                          | Unchanged Reused previously mapped design | 0h:00m:02s | 0h:00m:00s     | 165 MB           |
| decode                        | Unchanged Reused previously mapped design | 0h:00m:02s | 0h:00m:00s     | 164 MB           |
| io_buff                       | Unchanged Reused previously mapped design | 0h:00m:02s | 0h:00m:00s     | 164 MB           |
| regs                          | Unchanged Reused previously mapped design | 0h:00m:02s | 0h:00m:00s     | 164 MB           |
| work.eight_bit_uc.verilog     | Remapped Design changed                   | 0h:00m:02s | 0h:00m:00s     | 165 MB           |
| prgmcntr                      | Remapped Design changed                   | 0h:00m:02s | 0h:00m:01s     | 165 MB           |
| rom                           | Unchanged Reused previously mapped design | 0h:00m:02s | 0h:00m:00s     | 164 MB           |
| special_regs                  | Unchanged Reused previously mapped design | 0h:00m:02s | 0h:00m:00s     | 165 MB           |

## Clock Conversion and Distributed Synthesis

Clock conversions are handled the same during distributed synthesis and in a traditional (non-distributed) synthesis flow as follows:

- Clock conversion runs at the pre\_map stage, on compiler primitives.
- The names of referenced objects and the numeric values in the Clock Optimization report are defined with the conditions below:
  - Driving Element/Drive Element Type references compiler objects (e.g. MUX).
  - Clock trees with n number of instances and with the same inputs will be reported n times.
- ICG latch removal is performed in conjunction with clock conversion.
  - The ICG Latch Removal Summary is listed just before the Clock Optimization Report at the pre-map stage.
- ICGs that drive latches are supported and will be converted to registers.
- When the syn\_hier=hard attribute is applied to hierarchies that “cut” a gating cone of logic, the software does not prevent clock conversion to occur.

## Distributed Synthesis Limitations

Currently, distributed synthesis has some limitations:

- Does not support continue on error

- Does not support FSM Explorer
- The stability latch removal summary does not contain debug information for clock conversions.

# Analyzing Results

The following sections describe ways to analyze the results of a run command:

- [Checking Reports and Log Files](#), on page 512
- [Viewing Results in a Schematic](#), on page 517
- [Exploring a Schematic with find and expand](#), on page 520
- [Querying Incremental Results](#), on page 520
- [Querying Jobs](#), on page 523
- [Querying Metrics for a Design](#), on page 524
- [Dealing with Black Boxes in the Synthesized Netlist](#), on page 526
- [Checking Timing Results](#), on page 528
- [Handling Errors and Warnings](#), on page 550

## Checking Reports and Log Files

Each database generates its own reports, appropriate to the command used to generate that database state. The database is closed, so there are different ways to access the reports and log files. They are summarized in the table below; details are in the procedure that follows.

### Reports from the GUI (Within the database)

This is the recommended way to access the reports, as it is tightly integrated and includes features for navigation, locating errors, finding information about errors, and cross-probing. There are two ways to display the reports:

- Report View (step 1)
- view report and report list commands in the Tcl shell (step 2)

### Exported Reports Outside the Database

The export report command exports reports outside the database. This method is best suited for scripts (step 3).

### Reports Outside the Database (HTML)

It is recommended that you use the HTML version of the files if you need to view reports offline. It includes some of the support features provided by the GUI report (step 4).

## Reports Outside the Database (Text)

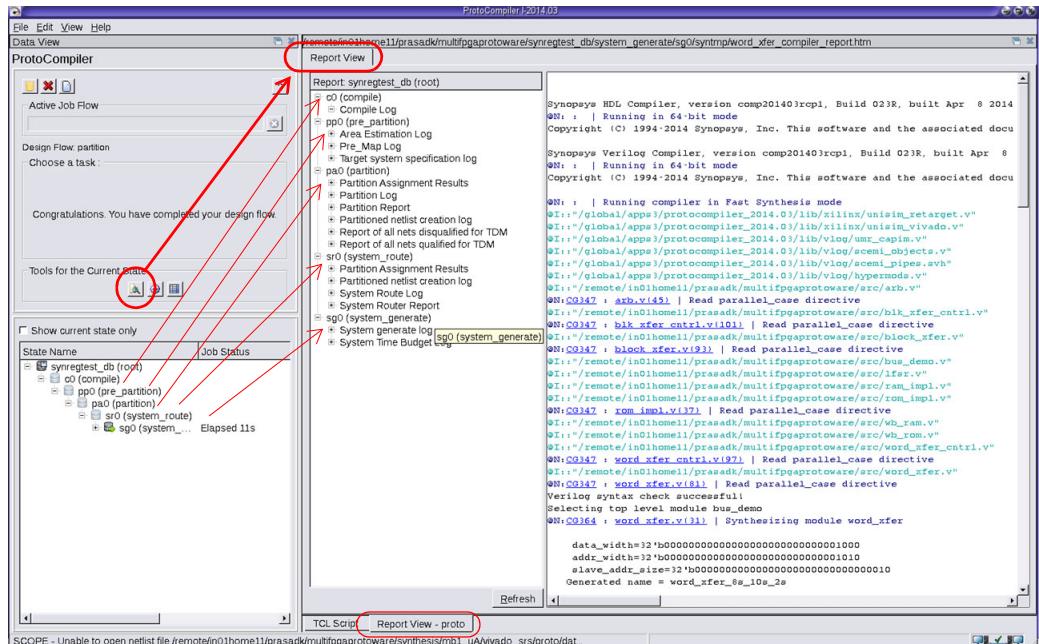
This version contains the same content, but does not have the navigation and other features available in the HTML or GUI versions of the report (step 5).

## Obfuscated Files that Cannot be Exported

Use the edit command from the GUI to view internal files that are referenced in messages but which cannot be exported (step 6).

- Follow the steps below to display the reports in the GUI Report View.
  - Go to the appropriate state in the database hierarchy.
  - To view all the reports up until that stage, click the Report View icon in the left panel in the tool GUI ( ) or type view report. This opens the Report view and displays the reports available for the active stage, as well as the reports for all its ancestors in the database tree. It does not display reports for any parallel branches.

In the following figure, the active state is the System Generate state, but the Report view displays the reports available for all the ancestors of this state. The reports are organized by database state.



- To display an individual report, use the list on the left of the Report view to navigate to the report you want.
2. To display reports in the GUI using the command line, do the following:
- To see a list of all available reports for a database state, use `report list`. This command lists the names of the reports in the Tcl console.
  - To open all the reports available from the current database back to the root, specify the `view report` command with no arguments.
  - To open a specific report, specify the `view report` command and the report name. The tool opens the specified report in the Report View.

The database is closed, so you must use these commands to access the reports and log files. See [Shell Command Reference, on page 17](#) in the *Command Reference Manual* for details about the commands.

3. To export a log file or report outside the database, use the `export report` command.
- Use `export report -list` to view the list of files available for export.
  - Use `export report -path` with the name of the file to export it to the specified location outside the database. Use the `-all` option to export all the available reports and log files.

This command is particularly useful in scripts.

4. To view reports offline, use the HTML version of the report.

It is recommended that you use this method of offline log viewing if you want to view results without opening the database. The GUI version of the reports described in the previous steps is part of the closed database.

- From outside the tool, open the HTML report file, which is a single-page summary of all the log files and reports with links to each one. It is located outside the database hierarchy, at the same level as the database, in the same directory. You can access all reports from the current database state back to the root. The reports displayed are the same reports as are available from the GUI, but they can be accessed from outside the database.
- The HTML report is not saved, but is overwritten by results from a subsequent run. To keep a report, save it to another location or another name.

5. To view a text version of a report offline, follow these steps:

- By default, the tool automatically generates text versions of the report files and puts them in a parallel directory to the database directory. If you do not see this directory, check that the `auto_export_reports` option is set to 1, using the `option set` command to reset it if necessary.
  - Go to the directory and open the reports you want.
  - These reports are overwritten by subsequent runs, so save any reports you want to keep to another location.
6. To display obfuscated files from the closed database that are referenced in messages but cannot be exported, follow the procedure below:

Although the `edit` command is also available from the command line, you can only open and edit obfuscated files from the GUI using the method described here.

- Locate the plain text file name referenced in the log file message. For example:

```
Error in subprocess: See log file /myDesign/results/EMAC/db0/pre_partition/pp2/synlog/addrcheck_netlist_optimizer.srr
```

- Open the database in the tool GUI. The database is required to be open for the `edit` command to open the file.
- Use the `edit` command in the Tcl window to open the file. You can then save the file to another location if you want.

```
edit /myDesign/results/EMAC/db0/pre_partition/pp2/synlog/addrcheck_netlist_optimizer.srr
```

7. Once you have opened the report you want, check the file.

The report typically lists the details and results of the current operation. The GUI version of the report includes color-coded errors, warnings and notes. See [Handling Errors and Warnings, on page 550](#) for information about working with these messages.

```

Report: DEMO (root)
#Build: HAPS® ProtoWare I-2014.09, Build 1780, Mar 6 2014
#Email: Y:\protoware\j301409_0510
#OS: Windows 7 6.1
#Hostname: RITA-6460P

Start of Compile
#Thu Mar 06 11:13:11 2014

Synopsys HDL Compiler, version compdewb, Build 3915R, built Mar 6 2014
#N : | Running in 64-bit mode
Copyright (C) 1994-2014 Synopsys, Inc. This software and the associated documentation are proprietary.

Synopsys Verilog Compiler, version compdewb, Build 8915R, built Mar 6 2014
#N : | Running in 64-bit mode
Copyright (C) 1994-2014 Synopsys, Inc. This software and the associated documentation are proprietary.

#N : | Running compiler in Fast Synthesis mode
#I:"Y:\protoware\201409_0210\lib\xilinx\unisim_retarget.v"
#I:"Y:\protoware\201409_0210\lib\xilinx\unisim_vivado.v"
#I:"Y:\protoware\201409_0210\lib\vlog\vhdl_capim.v"
#I:"Y:\protoware\201409_0210\lib\vlog\vhdl_objects.v"
#I:"Y:\protoware\201409_0210\lib\vlog\vhdl_semem_pipes.v"
#I:"Y:\protoware\201409_0210\lib\vlog\vhdl_semem_pipes.vvh"
#I:"Y:\protoware\201409_0210\examples\hpl_examples\src_demo_synthesis_flow\top.v"
#I:"Y:\protoware\201409_0210\examples\hpl_examples\src_demo_synthesis_flow\sub.v"
Verilog syntax check successful!
Relaunching top level module top
#N:QD841 : topo.v(11) | Synthesizing module top

#N:Q81160 : pop.v(4) | Found complex event expression
SEND

At c_ver Exit (Real Time elapsed 0h:00m:01s; CPU Time elapsed 0h:00m:00s; Memory used current: 8MB
Process took 0h:00m:01s realtime, 0h:00m:01s cputime
Thu Mar 06 11:13:13 2014

#####
#N : | Running in 64-bit mode
At sym_mfilter Kinit (Real Time elapsed 0h:00m:00s; CPU Time elapsed 0h:00m:00s; Memory used current: 8MB
Process took 0h:00m:01s realtime, 0h:00m:01s cputime
Thu Mar 06 11:13:14 2014

#####
SEND

```

## 8. Navigate to view specific pieces of information.

- In the GUI version, use the panel on the left to navigate to the section you want. In the HTML version, use the links.
- To search the body of the report, use Control-f to open a dialog box for search criteria.

The areas of the log file that are most important are the warning messages and the timing report, which lists the most critical paths. The following table lists common starting points for searches in the log file.

| To find ...             | Search for ...                     |
|-------------------------|------------------------------------|
| Notes                   | @N or look for blue text           |
| Warnings and errors     | @W and @E or look for colored text |
| Performance information | Performance Summary                |

| To find ...                                                              | Search for ...                                                              |
|--------------------------------------------------------------------------|-----------------------------------------------------------------------------|
| The beginning of the timing report                                       | START TIMING REPORT. Also see <i>Checking Timing Results</i> , on page 528. |
| Detailed information about slack times, constraints, arrival times, etc. | Interface Information                                                       |
| Resource usage                                                           | Resource Usage Report                                                       |
| Gated clock conversions                                                  | Gated clock report                                                          |

## Viewing Results in a Schematic

Schematics present graphic views of your design. The tool includes the HDL Analyst tool, where you can graphically view the design at different stages, as it progresses through the sequence of database states.

### HDL Analyst Versions

There are two versions of the HDL Analyst tool, the standard version, and the newer Beta version, which differ from each other.

See the following for details about each one in the *Compiler and Mapper Guide*:

| HDL Analyst (New)                                                 | HDL Analyst (Standard)                                                      |
|-------------------------------------------------------------------|-----------------------------------------------------------------------------|
| <a href="#">Analyzing With the HDL Analyst Tool</a> , on page 132 | <a href="#">Analyzing With the Standard HDL Analyst Tool</a> , on page 190  |
| <a href="#">Working in the Schematic</a> , on page 86             | <a href="#">Working in the Standard HDL Analyst Schematic</a> , on page 157 |
| <a href="#">Exploring Design Hierarchy</a> , on page 106          | <a href="#">Exploring Design Hierarchy (Standard)</a> , on page 166         |
| <a href="#">Finding Objects</a> , on page 113                     | <a href="#">Finding Objects (Standard)</a> , on page 172                    |
| <a href="#">Crossprobing</a> , on page 124                        | <a href="#">Crossprobing (Standard)</a> , on page 183                       |

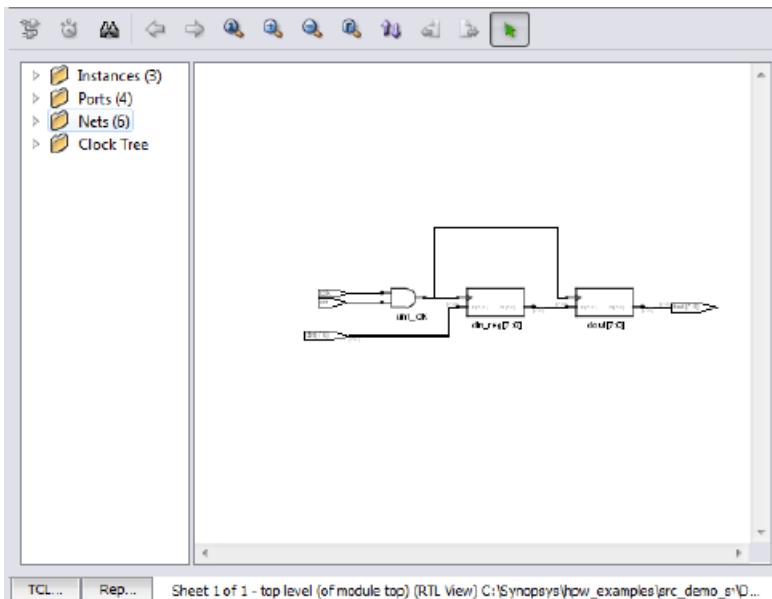
## Working in the Schematic

The following steps summarize how to analyze results in a schematic. For details about operations mentioned, refer to the appropriate section from the table above.

1. Type `view schematic` in the tool Tcl console, or click the schematic icon in the left panel of the tool GUI (  ) for GUI mode.

You can use the `view schematic` command in either GUI or shell mode, but GUI mode offers more functionality. In shell mode, the command does not open a schematic of the netlist, but enables the use of query and analysis commands like `find` and `expand`.

If you run the tool in GUI mode, the tool opens an HDL Analyst window with a graphic view of your design that reflects the state of the design at the given database state: for example, compiled, partitioned, mapped, routed, post-place-and-route, and so on. Use the `database set_state` command to change to the database state you want. You can use the object query commands and other HDL Analyst commands in this window to visually analyze the design.



There is a difference in memory usage, depending on the mode you use. In GUI mode, the memory associated with the design is released back to

the system when the HDL Analyst window is closed; in shell mode, the memory is released when the database is closed.

2. In GUI mode, use these techniques to navigate design hierarchy in the HDL Analyst view:
  - Use the browser on the left side, which lists objects by type.
  - Push down into a hierarchical object by placing the pointer over it and clicking. To go up a level, click in a blank area of the design.
  - If it is a multi-sheet schematic, move from one sheet to the next by clicking the appropriate icons.
3. In either GUI mode or shell mode, analyze the data in the schematic, using the techniques described below:

| Operation                                  | Action                                                                                                                                                                                                                                                                                                                                                                               |
|--------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Find objects                               | Use the Tcl <code>find</code> command to locate objects. See the next step for details.                                                                                                                                                                                                                                                                                              |
| View connected objects                     | Select an object as a starting point and then filter the design.                                                                                                                                                                                                                                                                                                                     |
| Expand logic                               | Select a pin on the object or a net; then right-click and select the <code>Expand</code> command that is appropriate. In shell mode, use the <code>expand</code> command.                                                                                                                                                                                                            |
| Filter the objects displayed               | Use any of these methods: <ul style="list-style-type: none"><li>• Select the objects you want to isolate and click F12 or the Filter Schematic icon. Click the Back icon to return to the previous view.</li><li>• Identify and select objects by type, using the browser on the left side of the view.</li><li>• Use the Tcl <code>find</code> command to locate objects.</li></ul> |
| Flatten the hierarchy                      | Right-click and select <code>Flatten Schematic</code> .                                                                                                                                                                                                                                                                                                                              |
| Display object properties in the schematic | <ul style="list-style-type: none"><li>• To temporarily display properties, hover over an object. A tooltip shows the information.</li><li>• Select an object, right-click, and select <code>Properties</code>. The properties and their values are displayed in a table.</li></ul>                                                                                                   |

For more information about the commands mentioned, see [Tcl Find and Expand, on page 371](#) in the *Command Reference Manual*. For usage

information, see [Finding Objects, on page 113](#) or [Finding Objects \(Standard\), on page 172](#) in the *Compiler and Mapper Guide*.

The commands apply to the current schematic, and remain relevant for that database state until you either close that database or issue another view schematic command for a different database state. If you are using GUI mode, the commands additionally require the HDL Analyst window to be open in order to take effect.

## Exploring a Schematic with find and expand

You can use object query commands like `find` and `expand`, and collection commands like `c_diff` to explore and analyze a design even if you are not using GUI mode.

1. From the database state you want to analyze, type `view schematic` in the shell window.

This command enables you to use the query and analysis commands in the next step. For more information, see [Viewing Results in a Schematic, on page 517](#).

2. Use the `find`, `expand`, or other commands to explore the database.

For details about the command, its syntax and examples, see [Tcl Find and Expand, on page 371](#) in the *Command Reference Manual*.

Depending on the HDL Analyst version you use, you can find additional usage information in [Finding Objects, on page 113](#) or [Finding Objects \(Standard\), on page 172](#) in the *Compiler and Mapper Guide*.

## Querying Incremental Results

Prototyping is iterative, and involves ongoing changes, because of design evolution and different versions, tweaks to optimize performance or address RTL changes, or debug instrumentation changes. Use the database `query_state` commands to get useful information about incremental results for a particular database state. For the complete syntax for this command, refer to [database, on page 30](#) in the *Command Reference Manual*. The following steps highlight some ways to use this command to extract useful information after a run:

1. Go to the database state that you want to query.

The command works on the current database state.

2. To use this command to get information about source files, do the following:

Specify the database query\_state command with the appropriate argument:

`database query_state argument`

| To view ...              | database query_state Argument |
|--------------------------|-------------------------------|
| A list of HDL 'defines   | -hdl_define                   |
| A list of HDL parameters | -hdl_param                    |
| The top module name      | -top_module                   |

3. To query options for a state, use the database query\_state -option command.

- To find the value set for a particular option, use this command:

`database query_state -option optionName`

- To print the settings for multiple options for the database state in readable text, include the database query\_state command in a Tcl command like this:

```
foreach o [option list] {puts "The value of $o when this state was
generated: '[database query_state -option $o']'}
```

- Alternatively, use the database query\_state -option command in a simple script, as shown in this example:

```
proc get_present_option_settings {} {
 set state_type [database query_state -run_type]
 set state_name [database get_state]
 puts ""
 puts "Option settings for state \"$state_name\" which is a state
 of type \"$state_type\""
 puts ""
 puts "Option Current Value"
 puts "-----"
 foreach o [option list] {puts "[format \"%-33s\" $o] '[database
 query_state -option $o]' " }
}
```

One good practice is to generate the list of option settings when a database state is first generated, before you begin experimenting with different values.

- To restore option settings to the values they had when you generated the database state, with a command like this one:

```
foreach o [option list] { option set $o [database query_state -option $o] }
```

4. To check whether a database state needs to be rerun, use the database query\_state -out\_of\_date command.

Use this command to check if the current state or states in the currently active tree are out of date and need to be rerun. This could be because input files or global options have changed, because an earlier state was rerun more recently than the current state, or because an earlier state was rerun with a different version of the software. For example:

```
% database query -out_of_date
pm0 is out of date:
 master.fdc has changed
m0 is out of date:
 Its parent pm0 is out of date
```

Use the -verbose option to see more details. You can also generate a report with this information by using the report out\_of\_date -mode command.

5. To check database state inputs, use the following commands:

| To...                                             | Commands                                              |
|---------------------------------------------------|-------------------------------------------------------|
| Compare inputs for current state to another state | database query_state -state_diff<br>report state_diff |
| List all inputs for current state                 | database query_state -input_files                     |

With the -state\_diff option, the command returns a 1 if the inputs are the same, and 0 if they are different. The equivalent report command generates a report with the comparison information for the specified states or trees.

The following example shows the results of comparing two mapper runs where changes were made for the second run (m1):

m0 differs with respect to m1:

arguments have changed  
my.fdc has changed

0

6. To generate reports about incremental changes use the following commands:

```
report out_of_date
report state_diff
```

These commands are similar to the corresponding database query\_state commands, but generate report files. For the complete syntax for the report commands, refer to [report out\\_of\\_date, on page 115](#) and [report state\\_diff, on page 121](#) of the *Command Reference Manual*.

## Querying Jobs

1. Use the job command to query pending, running, or completed jobs.
  - Use the job list command to list the jobs and their IDs.
  - Use the job query command to query a particular job, using its ID. For example:

```
job query job232 -run_time
11
```

See [job, on page 68](#) in the *Command Reference Manual* for details about the syntax of this command.

2. You can also create a simple Tcl script.

This example lists all the job IDs and job names, and reports their status.

```
set aa [job list -all];
puts "Job ID\t Job Name\t\tJob Status";
foreach ele $aa {
 puts [format "%-12s%-22s%5d" $ele [job query $ele -job_name]
 [job query $ele -return_code]]
}
```

## Querying Metrics for a Design

Keeping track of metrics is important for measuring and tuning the QoR of a design. Metrics include data from various steps in the design flow; the data is saved and can be retrieved anytime. The design metrics you can query include the number of LUTs, the runtime for each process step, the worst slack, the slack for specific clocks, and the number of unconverted gated clocks.

1. Start from the tool Tcl window.
2. To find the names of the metrics available for the design, use one of the following arguments with the `database query_state` command:

**database query\_state   Use it to...**

---

|                                 |                                                                                               |
|---------------------------------|-----------------------------------------------------------------------------------------------|
| <code>-dump_metrics</code>      | Shows the metrics that can be queried for the design.                                         |
| <code>-available_metrics</code> | Show metrics that can be queried for the design as a Tcl list that can be used for scripting. |

---

For details about the command syntax, see [database query\\_state](#), on page 35 of the *Command Reference*.

3. Use the metric names with one of the following `database query_state` commands, according to the level of detail you want to see.

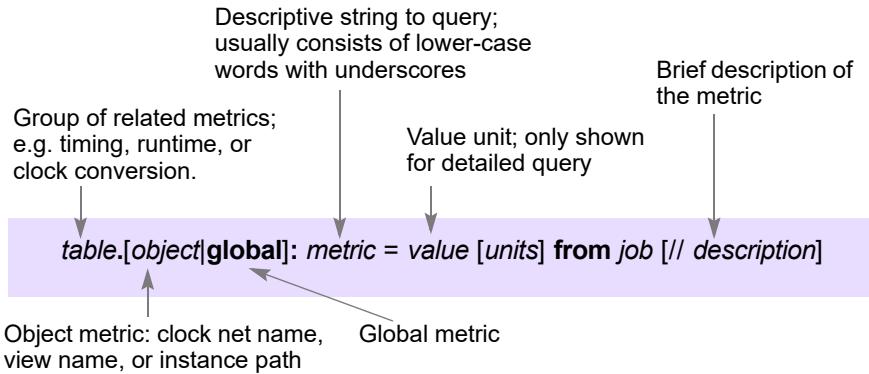
**database query\_state   Use it to ...**

---

|                              |                                                                                                                                                                                                                                                          |
|------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-metric</code>         | Query information about a QoR metric. For example:<br>% database query_state -metric<br>runtime.realtime -jobname compiler<br><br>3.154000                                                                                                               |
| <code>-metric_details</code> | Show metrics and values available for the current database.<br>% database query_state -metric_details<br>clock_conversion.clean_clock_pins -jobname<br>fpga_mapper<br><br>271 {} {Number of clock pins driven by<br>non-gated/non-generated clock trees} |

---

Metrics can be global for the entire design, or specific to an object, such as a clock, module, or net. The command returns values with the default output format shown below:



This is an example of the main fields for clock conversion metrics:

| <b>Table Value</b> | <b>Metric Name</b>    | <b>Description</b>                                                 |
|--------------------|-----------------------|--------------------------------------------------------------------|
| clock_conversion   | clean_clock_trees     | Number of non-gated/non-generated clock trees                      |
| clock_conversion   | clean_clock_pins      | Number of clock pins driven by non-gated/non-generated clock trees |
| clock_conversion   | gated_clock_trees     | Number of gated/generated clock trees                              |
| clock_conversion   | instancesConverted    | Number of sequential instances converted                           |
| clock_conversion   | instancesNotconverted | Number of sequential instances left unconverted                    |

This is an example of reported clock conversion metrics:

```
clock_conversion.global: instancesConverted = 0 from fpga_mapper
//Number of sequential instances converted

runtime.global: realtime = 4.160000 seconds from compiler
runtime.global: cputime = 3.073220 seconds from compiler
```

## Dealing with Black Boxes in the Synthesized Netlist

Black boxes in the synthesized netlist can cause problems during place and route. If you have black boxes, first identify why the information was missing for the black boxes, and then handle the situation as needed.

1. Identify the reason why the black boxes were created. There are two main reasons for this:
  - In designs that include DesignWare® IP, black boxes can be created for the IP if the appropriate tools can not be accessed. See step 2 for information about dealing with these black boxes.
  - In mixed HDL designs, the tool creates black boxes when it cannot identify entities because of mismatches. See step 3 for information about dealing with these black boxes.

These are some examples of possible messages:

CG389 @E: Reference to undefined module.

CD280 @W: Unbound component <name> mapped to black box.

2. To eliminate DesignWare IP black boxes, check that you have followed the guidelines below.

Not following these guidelines could cause the tool to generate black boxes for the IP.

- Run synthesis on a Linux machine.
  - Ensure that you have licenses for both DesignWare and Design Compiler®.
  - Make sure to use a compatible version of DesignWare; check the release notes for compatible versions.
  - Make sure you have correctly specified the path to the Design Compiler installation.
  - Make sure you have specified the DesignWare libraries appropriate to the IP you are using.
  - Provide the black box definitions before running place and route.
3. For black boxes in mixed-language designs, check the following:
    - Check for warnings related to port mismatches or missing module definitions.
    - Make sure VHDL entities have been correctly identified.

- Add the definition for the module or entity to the source file list before compiling. The definition can be in Verilog or VHDL format, or an EDIF or NGC netlist.

# Checking Timing Results

You can generate timing reports at different design stages to check the performance. The timing reports are based on estimates. The more mature the design, the more accurate the timing estimates reported.

See the following for more information:

- [Generating and Viewing Timing Reports](#), on page 528
- [Viewing and Correlating Results with the Timing Report View](#), on page 535
- [Generating Custom Timing Reports with the Timing Analyst](#), on page 541
- [Checking Clock Information](#), on page 546
- [Interpreting Gated Clock Error Messages](#), on page 895
- [Running System-Level Timing Analysis \(SLTA\)](#), on page 547

## Generating and Viewing Timing Reports

### Inter-FPGA Path Timing Report

Generate an inter-FPGA path timing report after System Route for timing analysis of the most critical nets crossing FPGA boundaries including multi-hop paths per unique start and end clock event pair. The report can aid in identifying what is causing major slacks and you can make changes accordingly to partition-routing and clock constraints. By fixing the timing on required paths, you will avoid multiple runs of runtime-intensive System Generate, Synthesis and Place & Route. After making appropriate changes, re-run from Pre-Partition, Partition, or System Route states until a satisfactory inter-FPGA path timing report is achieved.

To generate the report do the following after System Route.

```
report timing -generate -fdc fileName.fdc
```

or

```
report timing -generate -fdclist fileList
```

The report will be generated in the directory, *databasename\_reports/system-route/tim\*\_inter\_fpga\_path.rpt*

The complete syntax for the report timing -generate command is described in [report timing, on page 127](#) in the *Command Reference Manual*.

## Timing Report for Estimates of Design Performance

The following procedure describes how to generate a timing report with estimates of design performance. You can run intra-FPGA timing analysis on a single FPGA or inter-FPGA timing analysis in a multi-FPGA design.

1. To run timing analysis and generate a timing report, use this basic command:

```
report timing -generate
```

The tool runs timing analysis on the current database (mapped or partitioned netlist) and generates a report file, which is displayed automatically. The report has a .ta.rpt extension. The timing report varies, based on the netlist analyzed.

The complete syntax for the report timing -generate command is described in [report timing, on page 127](#) in the *Command Reference Manual*.

2. Use the -mode argument to specify the netlist you want to analyze and correspondingly, the type of timing analysis you want to run:

| Timing Report Type                                                                                                                    | Netlist Type                         | -mode Option           |
|---------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------|------------------------|
| Single-FPGA. Command run from mapped database                                                                                         |                                      |                        |
| Post-synthesis                                                                                                                        | Mapped                               | None, or -mode default |
| With backannotated place-and-route data                                                                                               | Mapped, with backannotated data      | -mode ba               |
| Multi-FPGA. Command run from system generate database. See <a href="#">Running System-Level Timing Analysis (SLTA), on page 547</a> . |                                      |                        |
| Post-partitioning                                                                                                                     | Partitioned                          | None, or -mode default |
| With backannotated place-and-route data for FPGAs                                                                                     | Partitioned, with backannotated data | -mode slta             |

To use backannotated data, you must run place and route and import the backannotated data before generating the report. You can then run a command like this one that is issued from the mapped state:

```
report timing -generate -mode ba
```

If you do not specify **-mode**, timing analysis uses the netlist for the current map or system generate database state as the default.

3. To customize the timing report, do the following:

- Open a schematic from a particular database state with the **view schematic** (map database state) or **view schematic time\_budget** (system generate state) command. You must explicitly do this; if not, the tool opens the compiled schematic, which does not use post-mapping names for the objects. See *Identifying Objects Correctly in Post-Mapped Timing Reports*, on page 533 for more information about naming issues.
- Narrow down the results by specifying the arguments you want when you run the command.

| To Report Paths ...                                         | Command Arguments                                                            |
|-------------------------------------------------------------|------------------------------------------------------------------------------|
| To/from/through specified ports, registers, pins, or clocks | <b>-from</b><br><b>-to</b><br><b>-through</b>                                |
| Based on slack times                                        | <b>-slack margin</b>                                                         |
| Based on rise/fall clock edges                              | <b>-fall_from</b><br><b>-fall_to</b><br><b>-rise_from</b><br><b>-rise_to</b> |

- To make the results even more specific, use object query commands in combination with the from/to/through options listed above. For example:

```
report timing -generate -through [get_pins {ram_to_bus_Q_2_1.O}]
report timing -generate -through [get_pins {ram_to_bus_Q_2_1.O}] -mode ba
```

See *Object Query Commands*, on page 403 in the *Command Reference Manual* for the syntax of these commands. If you use the object query syntax and run the **report timing -generate** command from within a script, make sure to open the schematic first, so that the specified objects can be located.

4. To keep previous timing reports, use the `-out` option with `report timing -generate` to create multiple reports for a particular database state.

You can run multiple timing analysis runs on the same netlist and from the same database state. By default, the current timing report overwrites a previous one. When you specify a name with the `-out` option, the tool creates a substate under the current database state with the name you specified. It then generates a timing report for the substate, called `stateName.substateName.ta.rpt`. For example, if you specify `-out trial1` from the `m0` database state, the timing report is called `m0.trial1.ta.rpt`.

5. Check the timing results by viewing the report.

See [Checking Reports and Log Files, on page 512](#) for tips on working with reports.

6. Use the `check_hstdm_timing` utility to generate a report where you can check that all HSTDMD paths are covered by DPO (data path only) exception constraints and that there are no missing timing budget constraints.

Common reasons for missing budget constraints are GCC issues, inferred clocks, differences in sequential optimization between time budgeting and synthesis, and IP synthesis issues.

- After mapping, run the `check_hstdm_timing` utility using the options you need to identify missing constraints. Check that all HSTDMD paths are covered by DPO timing exceptions. To save runtime, specify the options you need; otherwise the utility runs all the reporting options. For details about the utility, see [check\\_hstdm\\_timing Utility Syntax, on page 532](#).
- Specify the `-rx2user` and `-user2tx` options with `check_hstdm_timing`. The paths from receiver to user logic and from user to transmit logic are the ones that most commonly have missing constraints, so use these corresponding options to identify them.

The utility generates a timing report: `dir/timing_correlation_dbStateName/fpgaHstdmTiming.rpt`. The report is a table that shows start and end point details.

- If the Constraints column in the report shows None, it indicates an error.

Paths that go through HSTDMD points in SLTA actually start or end in HSTDMD logic for FPGA synthesis. The tool handles this by generating

fdc time budget files with data path only (DPO) constraints like the ones shown in the following example.

```
Time budget constraints from user logic to HSTDMD logic (transmit)
Constraints for Ratio 4
set cpm_snd_HSTDMD_4by2_fb1_B23_A_4_data_0_c_pciclk_r [expr 46.5
 + $HSTDMD_4by2_snd_constraint_adjust]
set_datapathonly_delay -comment {HSTDMD tx budget HSTDMD_4by2
 (fb1.uB:J23)->(fb1.uC:J13)} -rise_from [get_clocks
 -include_generated_clocks {c:pciclk}] -to
 {t:cpm_snd_HSTDMD_4by2_fb1_B23_A_4.bit_tx.mserdes.D1}
 $cpm_snd_HSTDMD_4by2_fb1_B23_A_4_data_0_c_pciclk_r

Time budget constraints from HSTDMD logic to user logic (receive) |
Constraints for Ratio 4
set cpm_rcv_HSTDMD_4by2_fb1_B23_A_4_data_0_c_pciclk_r [expr 26.1
 + $HSTDMD_4by2_rcv_constraint_adjust]
set_datapathonly_delay -comment {HSTDMD rx budget HSTDMD_4by2
 (fb1.uB:J23)->(fb1.uC:J13)} -from
 {i:cpm_rcv_HSTDMD_4by2_fb1_B23_A_4.rxbit.ISERDES_RX_DATA_00}
 -rise_to [get_clocks -include_generated_clocks {c:pciclk}]
 $cpm_rcv_HSTDMD_4by2_fb1_B23_A_4_data_0_c_pciclk_r
```

- Look for inferred clocks and declare them.
7. Write out the missing DPO constraints in the FDC and XDC files by running `check_hstdm_timing` with the appropriate options.
- Use the `-default_delay` option to specify a DPO delay value to use when writing out the missing constraints; otherwise the default value is 8. Base the DPO value on the TDM ratio, using a higher value for a higher ratio. Check the relative values from DPO categories that are not missing to get an idea.
  - Run the utility with the `-write_fdc` and `-write_xdc file` options, to write out the missing constraints in the FDC and XDC files, respectively.
  - Check the constraints in the xdc file and add the file to the Vivado run when you place and route.

## **check\_hstdm\_timing Utility Syntax**

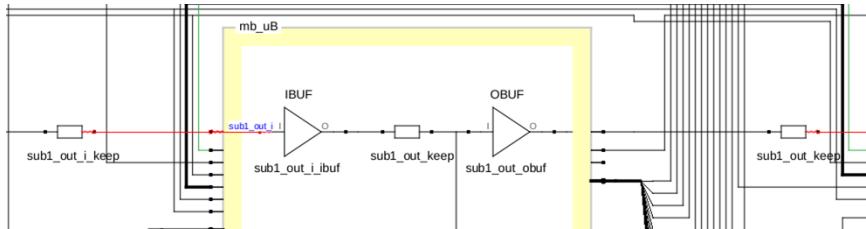
The `check_hstdm_timing` utility is a Tcl utility that identifies missing budget constraints and also writes out the missing constraints to the xdc and fdc files. Run it from the tool Tcl window after mapping. The utility has optional arguments; if it is run without any options, the utility takes longer because it runs all the reporting options.

| Argument                          | Description                                                                                           |
|-----------------------------------|-------------------------------------------------------------------------------------------------------|
| <code>-default_delay value</code> | Specifies the DPO delay value to use when writing missing constraints. The default is 8.              |
| <code>-port2tx</code>             | Reports paths from input ports to the transmit logic.                                                 |
| <code>-rx2port</code>             | Reports paths from the receive logic to output ports.                                                 |
| <code>-rx2tx</code>               | Reports paths from the receive logic to the transmit logic (multi-hop).                               |
| <code>-rx2umr</code>              | Reports paths from the receive logic to the UMR logic.                                                |
| <code>-rx2user</code>             | Reports paths from the receive logic to user logic.                                                   |
| <code>-slack value</code>         | Reports only those paths with a slack less than the given value. Default is 1500 ns.                  |
| <code>-umr2rx</code>              | Reports paths from the UMR logic to the receive logic.                                                |
| <code>-user2tx</code>             | Reports paths from user logic to the transmit logic.                                                  |
| <code>-write_fdc</code>           | Writes out an FDC file with DPO constraints for paths that were missing timing exception constraints. |
| <code>-write_xdc fileName</code>  | Adds missing DPO constraints to the specified XDC file. You must specify the file name.               |

## Identifying Objects Correctly in Post-Mapped Timing Reports

When timing reports are based on mapped databases, object names might no longer match the original RTL names. For example a RAM that was `{i.dest_u4.ram1[7:0]}` in the RTL might be called `{i:dest_u4_ram1_ram1_0_0}` in the mapped database. When you generate customized timing reports, you cannot use the RTL name of the object; the name must match the mapped name.

Nets with `-through` timing constraints can get segmented, and parts of the original constraint might no longer be applicable for SLTA. In the following figure, because of segmentation, the original false path constraint no longer applies to the left part of the net shown. However it still applies to the segment of the net on the right.



To make sure you identify objects correctly, do the following:

1. Open the schematic of the design with the `view schematic` command (map database state) or `view schematic time_budget` command (system generate state).

This command opens a post-map view that has the correct names.

2. Use `fdc query` commands to identify the object correctly so that you can use it in your timing report generation command. For example:

```
% find -hier -inst {i:dest_u4.ram1*}
i:fb.uB.dest_u4.ram1_ram1_0_0
% report timing -generate -netlist slta -to [get_cells -hier
{i:dest_u4.ram1*}] -out SLTA
```

## Timing Report Example

This timing report is generated from the map stage with backannotated data (-mode ba). You can check location information in the report.

```

Top view: diff_clk_domains2
Requested Frequency: 100.0 MHz
Wire load mode: top
Paths requested: 1
through: ram_to_busQ_2_1_0
The system contains the following FPGAs:
Constraint File(s):
@N:MT320 : | Timing report estimates place and route data. Please look at the place and route timing report fo
 ↴

Worst From-To Path Information

Path information for path number 1:
 Requested Period: 10.000
 - Setup time: 0.000
 + Clock delay at ending point: 0.577
 = Required time: 10.577

 - Propagation time: 2.002
 - Clock delay at starting point: 2.307
 = Slack : 6.268

 Number of logic level(s): 2
 Starting point: RAM1.par[13] / Q
 Ending point: ram_to_busQ / D
 The start point is clocked by dcm_test1|CLK0_BUFS_derived_clock [rising] on pin C
 The end point is clocked by dcm_test1|CLK0_BUFS_derived_clock [rising] on pin C

Instance / Net Pin Pin Arrival No. of Location
Name Type Name Dir Delay Time Fan Out(s)

RAM1.par[13] FD Q Out 0.269 2.576 - (TILE_x227y258)
ram_par[13] Net - - 0.894 - 2 -
ram_to_busQ_2_1 LUT4 I1 In - 3.470 - (TILE_x227y258)
ram_to_busQ_2_1 LUT4 O Out 0.053 3.523 - (TILE_x227y258)
ram_to_busQ_2_1_i Net - - 0.344 - 1 -
ram_to_busQ_2 LUT6 I4 In - 3.867 - (TILE_x227y258)
ram_to_busQ_2 LUT6 O Out 0.106 3.973 - (TILE_x227y258)
ram_to_busQ Net - - 0.336 - 1 -
ram_to_busQ FD D In - 4.309 - (TILE_x225y258)
=====
Total path delay (propagation time + setup) of 2.002 is 0.428(21.4%) logic and 1.574(78.6%) route.
Path delay compensated for clock skew. Clock skew is added to clock-to-out value, and is subtracted from setup

```

## Viewing and Correlating Results with the Timing Report View

The Timing Report View provides a convenient way to view timing reports at certain stages of the design. It lets you view and query critical timing paths, and correlate certain timing results. You can view system-level timing, post-synthesis timing results alone, or in a composite view where they are correlated with the place-and-route timing results. See these topics for details:

- [Viewing System-Level Timing Results in the Timing Report View](#), on page 536
- [Viewing Synthesis Results in the Timing Report View](#), on page 537
- [Correlating Synthesis Results with Place-and-Route Timing](#), on page 539

## Viewing System-Level Timing Results in the Timing Report View

The following procedure describes how to use the Timing Report view to display and analyze system-level timing information.

1. Run system generate.

This generates the default system-level timing results.

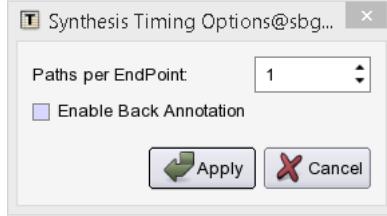
2. Display the Timing Report View using one of the methods described below.

- To display the default system-level timing results, go to the system generate state and select Tools->Timing Report View or click the Timing Report view icon (  ).

The Timing Report view opens (see [Timing Report View, on page 40](#) in the *Reference Manual*). By default, the view shows the system timing summary in the System Timing tab. The summary matches the critical paths shown in the Clock Relationships section of the log file after synthesis.

3. Set reporting and display options and analyze results.

- To expand all paths for the clocks and instances, click Expand All.
- To collapse all paths for the clocks and instances, click Collapse All.
- Click Options to open the Synthesis Timing Options dialog box. Set the number of paths to display in the Paths per EndPoint field. Check the Enable Back Annotation box to use back-annotated timing data.
- To sort data click on column headers.



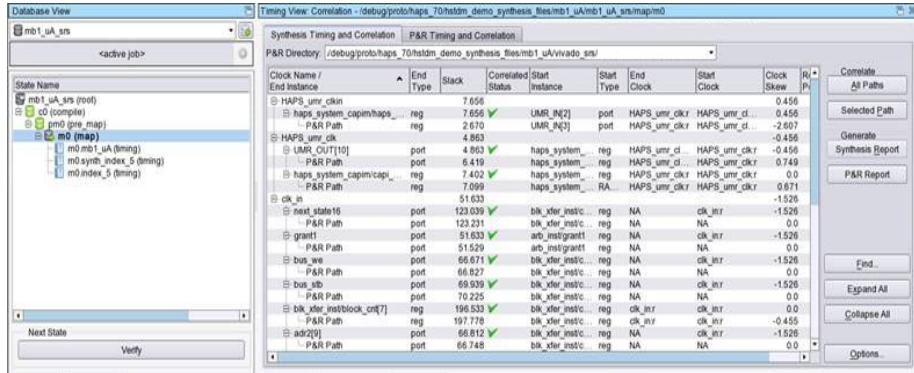
## Viewing Synthesis Results in the Timing Report View

The following procedure describes how to use the Timing Report view to analyze synthesis timing information from the Map state (Synthesis flow).

### 1. Display the Timing Report View.

- From a mapped view, select Tools->Timing Report View or click the Timing Report view icon ( ).

The Timing Report view opens (see [Timing Report View, on page 40](#) in the *Reference Manual*). By default, the view shows the synthesis timing summary in the Synthesis Timing and Correlation tab. The summary matches the critical paths shown in the Clock Relationships section of the log file after synthesis.

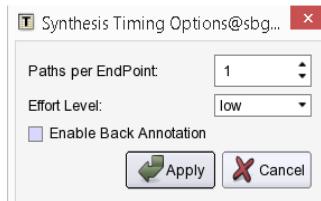


- Go to the Synthesis Timing and Correlation tab (default), if it is not open at this tab. The default timing view shows logic synthesis timing.

### 2. Set reporting and display options.

- To expand all paths for the clocks and instances, click Expand All.
- To collapse all paths for the clocks and instances, click Collapse All.

- To change the number of paths displayed per end point, select the path and click Options. Set the number of paths to display in the Synthesis Timing Options dialog box that opens.



- For the Synthesis Timing and Correlation that correlate to the place and route results, you can specify the following effort levels:
  - low - Runs a default timing report for Vivado (no -from/-to filtering). From this report a first-pass correlation is run. For paths that did not correlate, switch to effort Medium and rerun for All Paths, or select individual paths and run Selected Path.
  - medium - Performs a low effort run, but continues to run the second-pass correlation until there is no further convergence of results. This occurs when the number of correlation failures does not change from the previous run.
  - high - Performs a medium effort level run. However, whenever convergence stops, it then continues running correlation on the remaining individual paths until all have been attempted. Be aware that effort level high can incur large runtimes, because Vivado is called repeatedly.
- To locate objects like clocks or instances, click Find and type in the search term:



- To generate a timing report summary for a path, select the path and click Synthesis Report.

The screenshot shows the Synopsys HAPS Prototyping User Interface. The main window has two tabs: "Database View" and "Timing View Correlation". The "Timing View Correlation" tab is active, showing a table of timing analysis results. The table includes columns for Clock Name, End Instance, Type, Black, Correlated Start Status, Instance, Start, End, Type, Clock, Start Clock, Clock Skew, Req, and Period. A red arrow points to a specific row in the table.

| Clock Name /<br>End Instance | Type | Black | Correlated Start Status | Instance | Start | End          | Type | Clock      | Start Clock  | Clock Skew | Req   | Period |
|------------------------------|------|-------|-------------------------|----------|-------|--------------|------|------------|--------------|------------|-------|--------|
| NA @ MaxDelay Path 54.800n   | pof  |       |                         | 51529    | ✓     | arb_intgrant | reg  | NA         | NA           | -1.526     | 0.0   | 54.800 |
| > grant                      | pof  |       |                         | 51633    | ✓     | arb_intgrant | reg  | NA         | clk_inr      | -1.526     | 0.0   | 54.800 |
| > Synthesis                  | pof  |       |                         | 51772    | ✓     | haps_system  | reg  | HAPS_umr_d | HAPS_umr_ckr | -0.749     | 0.456 | 54.800 |
| > UMR_GOUT[5]                | pof  |       |                         | 4663     | ✓     | haps_system  | reg  | HAPS_umr_d | HAPS_umr_ckr | -0.456     | 0.456 | 54.800 |
| > Synthesis                  | pof  |       |                         | 2670     | ✓     | haps_system  | reg  | HAPS_umr_d | HAPS_umr_ckr | -2.607     | 2.607 | 54.800 |
| > haps_system_capiin.haps    | reg  |       |                         | 2670     | ✓     | UMR_IN[0]    | pof  | HAPS_umr_d | HAPS_umr_ckr | -2.607     | 2.607 | 54.800 |

The bottom half of the interface shows a terminal window with a log of command-line activity. The log includes messages about synthesis, place-and-route, and design completion, along with memory usage and clock conversion details.

```

1 # Tue Feb 14 11:58:00 2017
2
3 Synopsys Xilinx Technology Mapper, Version major: Build 97950, Build 97950, Built Feb 13 2017 09:47:14
4 Copyright (C) 1994-2017 Synopsys, Inc. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used
5 Product Version: M-2017.03
6
7 Mapper Startup Complete (Real Time elapsed 0h:00:00s; CPU Time elapsed 0h:00:00s; Memory used current: 98MB peak: 98MB)
8
9 9N: HF240 [Running in 64-bit mode]
10 9N: HF666 [Clock conversion enabled. (Command "set_option -fix_gate_and_generated_clocks 1" in the project file.)]
11
12 Design Input Complete (Real Time elapsed 0h:00:00s; CPU Time elapsed 0h:00:00s; Memory used current: 96MB peak: 101MB)
13
14
15 Mapper Initialization Complete (Real Time elapsed 0h:00:00s; CPU Time elapsed 0h:00:00s; Memory used current: 98MB peak: 101MB)
16
17
18 Start loading timing files (Real Time elapsed 0h:00:00s; CPU Time elapsed 0h:00:00s; Memory used current: 101MB peak: 101MB)
19
20
21 Finished loading timing files (Real Time elapsed 0h:00:00s; CPU Time elapsed 0h:00:00s; Memory used current: 101MB peak: 104MB)
22
23 Reading chip information from file <C:\Users\lindan\Desktop\tpga01703_329p\lib\xilinx\plandata\xcv440-flgs292>
24 9N: HF203 [Set auto_constraint_io
25 Adding property syn_ta_paths_and_display_worst_paths, value 1 to viewTraceBuild.tcl::top(verb)
26 9N: HF294 [Setting synthesis effect to very_low for the design
27
28 Start Timing Analyzer (Real Time elapsed 0h:00:05s; CPU Time elapsed 0h:00:05s; Memory used current: 289MB peak: 281MB)
29
30
31 Start final timing analysis (Real Time elapsed 0h:0m:07s; CPU Time elapsed 0h:00:06s; Memory used current: 289MB peak: 281MB)
32
33 9N: HT246 [Blackbox UMRGND_IF is missing a user supplied timing model. This may have a negative effect on timing analysis and optimizations (Quality
34 9N: HT246 [Blackbox UMRGND_IF is missing a user supplied timing model. This may have a negative effect on timing analysis and optimizations (Quality
35 9N: HT246 [Blackbox HPSI_GND is missing a user supplied timing model. This may have a negative effect on timing analysis and optimizations (Quality
36 9N: HT246 [Blackbox HPSI_GND is missing a user supplied timing model. This may have a negative effect on timing analysis and optimizations (Quality
37 9N: HT245 [Found clock HAPS_umr_ckr with period 10.00ns
38 9N: HT245 [Found clock HAPS_umr_ckr with period 10.00ns
39 9N: HT245 [Found clock HAPS_umr_ckr with period 10.00ns
40 9N: HT245 [Writing timing correlation to file C:\Users\lindan\Desktop\tpga01703_329p\bin\SYSTEM_IP_MIO_GPIO_E2121_DEMO_reports\system_generate.sgp]
41
42 9N: ##### START OF TIMING REPORT #####
43 # Timing Report written on Tue Feb 14 11:58:07 2017
44 #
45
46

```

## Correlating Synthesis Results with Place-and-Route Timing

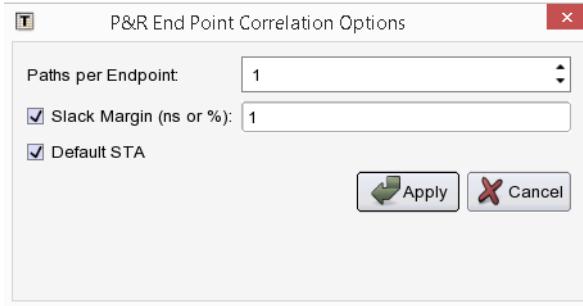
The following procedure describes how to use the Timing Report view to generate information from place-and-route timing, and compare it to the results from synthesis. Synthesis timing is considered the golden timing when correlating to P&R when in the Synthesis Timing and Correlation tab and P&R Timing results are considered golden when correlating to Synthesis when in the P&R Timing and Correlation tab. If you want to view the synthesis timing results only, use the procedure described in [Viewing Synthesis Results in the Timing Report View, on page 537](#).

- To generate place-and-route timing information, follow these steps:
  - Synthesize, place, and route the design. See [Running Place and Route without Exploration, on page 580](#) for the basic procedure.
  - Display the Timing Report View by selecting Analysis->Timing Report View or by clicking the Timing Report view icon (  ). By default the view displays the timing results after mapping.
- Set up to correlate results.
  - Select the P&R Timing and Correlation tab.

- In the P&R Directory field specify where the Vivado P&R results are located. Typically this is the directory specified with the `export vivado -path` command for the Vivado run.
3. Run correlation and analyze results.
- To correlate all paths, click the All Paths button on the right.
  - To only correlate selected paths, click Selected Path and select the paths you want.

The window displays the place-and-route timing information and correlates it to the synthesis information. For example, you can compare synthesis timing results with P&R Static Timing Analysis (STA) results. The view reports the status of end points, start points, and required periods. Paths are reported against the end clock. The P&R path is shown first, with the synthesis path to be correlated below.

- Look for red X marks ( in the Correlated Status column, which indicate a mismatch. A green check mark () means that the synthesis and P&R timing results for end points, start points, and requested periods on that path match.
  - Float over the mismatched cell for a tool tip that describes the error. For a complete description of the Timing Report View options, see [Timing Report View, on page 40](#) in the *Reference Manual*.
4. To view and analyze individual paths, use the following features in the view:
- To generate a detailed timing report for a particular path, select the path and click Synthesis Report.
  - To generate a detailed P&R timing report for a path, select the path and click P&R Report.
5. Set other reporting and display options as needed.
- Order the paths in ascending or descending order of slack values by sorting on the Slack column.
  - Change P&R timing correlation options by clicking the Options button and specifying the settings you want. For example, you can specify the number of paths to report per end point. If you do not enable any of the limiting options, the report includes paths for all the constraints. Rerun correlation by clicking All Paths or Selected Paths to view the updated results.



- To expand all paths for the clocks and instances, click Expand All.
- To collapse all paths for the clocks and instances, click Collapse All.
- To change the number of paths displayed per end point, select the path and click Options. Set the number of paths to display in the Synthesis Timing Options dialog box that opens.
- To locate objects like clocks or instances, click Find and type in the string to locate.

## Generating Custom Timing Reports with the Timing Analyst

You can generate a customized timing report (.ta) when you need more details about a specific path, or for a path that is not one of the top five timing paths (covered in the log file by default). To generate the custom timing report, the tool uses the built-in timing analyst engine.

You can generate a custom timing report from a system generate or mapped database state. From the mapped state, you can alternatively launch Vivado and use the place-and-route engine to generate the timing report.

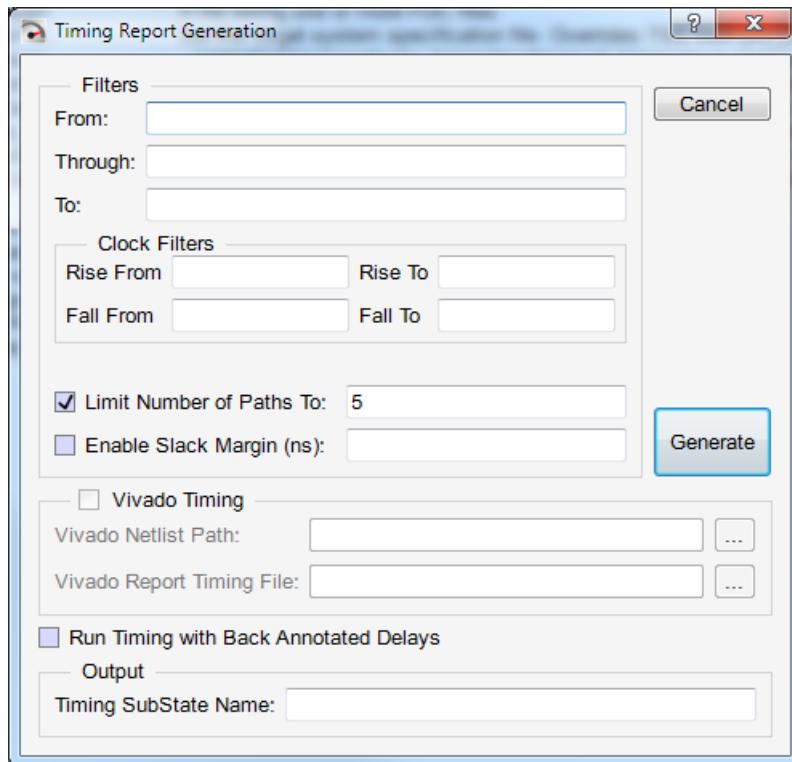
The following procedure shows you how to generate a custom report:

1. From a system generate or mapped database state, click the Timing Analyst icon ( ) or select Tool->Timing Analyst.

From a mapped state, you can run detailed timing analysis using either the Timing Analyzer or the Vivado timing engine. From a system generate state, the tool runs system-level timing analysis (SLTA).

2. In the dialog box that opens, use the From, Through, and To fields to filter and specify the paths you need.

- For timing analysis with the Timing Analyst, start from the map or timing budget schematic. Either cut and paste or drag and drop valid objects from there into the dialog box. Include the appropriate prefix if you type in the names: {i:u.1.q}. You can also use the get command to enter names.
- For Vivado timing analysis, enter names in the Vivado format with / as the hierarchy separator. If they are entered in the Synopsys format ({i:u.1.q}), the tool attempts to convert them automatically to the Vivado format: {u1/q}). You can also use the get\* query commands in these fields.
- Specify the objects using the guidelines described in *Specifying From, To, and Through Points for Custom Timing Reports*, on page 544.



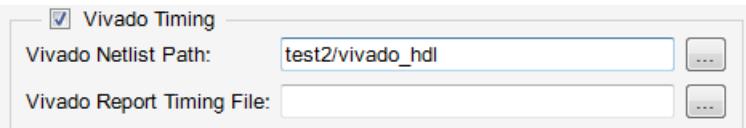
### 3. Specify other options for the custom timing report.

- For the rise and fall options in the Clock Filters section, make sure to enter clock aliases with the c: prefix.

For Vivado timing analysis, enter Vivado clock names. Get the names either from the Vivado place and route timing summary generated by the prototyping tool (*vivadoDir/\*\_timing\_summary.txt*) or from Vivado (*vivadoDir/vivado.log*).

You can also use the `get*` query command to filter the clocks. See [Specifying From, To, and Through Points for Custom Timing Reports, on page 544](#) for details.

- If needed, change the number of paths generated by setting the Limit Number of Paths to: and Enable Slack Margin options. If both are set, the report is based on the specification that is the most constricting.
  - Specify a name for the output timing report (ta).
4. To generate a custom timing report from the map state, click Generate. The tool generates a timing substate under the current database state, and a custom report file (.ta). You can also view the corresponding schematic for the timing paths.
5. To generate a custom timing report from the map state using the Vivado engine instead of the Timing Analyst, do the following:
- Start from a mapped state and run `export netlist` to create and set up the directory structure for place and route. Then use `launch vivado` to run place and route.
  - From the mapped state, start the Timing Analyzer and enter the filter criteria as described in steps 1 to 3. Use the Vivado format for names, with the / hierarchy separator, if possible. You can also use the `get` command to enter names: `[get_cells {u1/q}]`.
  - Enable Vivado Timing.



- In Vivado Netlist Path, point to the Vivado directory created by `export netlist`.
- Specify a name for the timing report file in Vivado Report Timing File. If no path is specified, it defaults to the current working directory.
- Click Generate.

The tool launches and runs Vivado timing analysis for the selected paths instead of the Timing Analyst.

6. To run system-level timing analysis, do the following:
  - Start from a system generate state.
  - Start the Timing Analyzer and enter the filter criteria as described in steps 1 to 3.
  - Click Generate.

The tool runs system-level timing analysis (SLTA). The timing analysis reports run from this state report inter-FPGA paths; to see detailed intra-FPGA timing, run timing analysis from a mapped state. For additional information about the SLTA process, see [Running System-Level Timing Analysis \(SLTA\), on page 547](#).

7. To run system-level timing analysis with backannotation, do the following:
  - Partition the design, and synthesize and place and route the FPGAs.
  - Backannotate the place and route results into each FPGA as described in [Importing Place and Route Results for Backannotation, on page 584](#).
  - Start the Timing Analyzer from a system generate state and enter the filter criteria as described in steps 1 to 3.
  - Enable Run Timing with Back Annotated Delays.



- Click Generate.

The tool runs SLTA using the back-annotated data.

8. Analyze results using the timing file and schematic.

## Specifying From, To, and Through Points for Custom Timing Reports

## Specifying Through Points

- Specify a through point by itself or in combination with From and To.
- Specify a net, hierarchical port, or instantiated cell pin. You can specify multiple objects. You can also enter a get\* query command.
- Typed-in names must include the appropriate n: or t: object prefix. Drag and drop automatically enters the syntax correctly. Synopsys-style names are automatically converted for place and route. To type in Vivado-style names for Vivado timing analysis, use the get command. For example: [get\_cells {u1/q}].
- You can use wildcards, as described in *Timing Analyst Examples with Wildcards , on page 55* in the *Reference Manual*.
- To generate a report for paths that go through ANY of the points specified (OR list), use a space-separated list that is optionally enclosed in curly braces. This example reports every path that passes through either port a12 or port c3:  
{t:a12 t:c3}

See *Path Filter Combinations for the Timing Analyst , on page 52* in the *Reference Manual* for examples.

## Specifying From/To Points

- Specify a port, register, register pin, or clock alias. You can specify multiple objects. You can also enter a get\* query command; for example: [get\_cells {u1/q}].
- Typed-in names must include the appropriate p: (top-level port), t:, (hierarchical port or instance pin) i: (instance), or c: (clock alias) object prefix. Drag and drop automatically enters the syntax correctly. Synopsys-style names are automatically converted for place and route.
- To type in Vivado-style names for Vivado timing analysis, use the get command. For example: [get\_cells {u1/q}]. See *Path Filter Combinations for the Timing Analyst , on page 52* in the *Reference Manual* for examples.
- You can use wildcards in the names, as described in *Timing Analyst Examples with Wildcards , on page 55* in the *Reference Manual*.
- Narrow down results by specifying both From and To, or specify just one of them.

## Specifying Clock Rise/Fall From/To Points

- Specify a clock alias as listed in the Performance Summary. Make sure to include the c: (clock alias) object prefix. You can also use the get command to specify names.
- To specify clocks for Vivado timing, use the clock names from the Vivado Place and Route Timing summary in the tool, or from the Vivado log. You can use the get command to specify names.
- You can use wildcards in the names, as described in *Timing Analyst Examples with Wildcards , on page 55* in the *Reference Manual*.

## Checking Clock Information

You can check clock information at different points in the prototyping process.

1. Check the clock information in the log file (view report command or Report View icon) at any stage of the design.

This lists the reports available for the current database state.

2. After compile, check for constraint errors with run report constraint\_check.

The command generates a report that lists errors in constraints. Fix the errors before proceeding.

3. Check the gated clock report for information about gated clock conversion.

See [Interpreting Gated Clock Error Messages, on page 895](#) for information about specific messages.

4. Check clock information after pre-partition.

- Run report target\_system to generate the reports. Use the view report pcf\_tss\_report.rpt command to view the TSS report, which includes details about the target system.
- Check that all clocks are listed in the Clock Summary section.
- Check that all clocks have been declared. If there are any inferred clocks, specify clock definitions for them with FDC create\_clock constraints.
- Check the values in the Clock Settings section of the TSS report.
- Check for inapplicable constraints and fix them before proceeding.

5. Check clock skew.

- After partitioning, check the log file for clocks and asynchronous resets that cross FPGA boundaries.
- Specify clock tree replication and rerun partition:

```
run partition -clock_gate_replicaton 1
```

For details, see [Partitioning Clocks, on page 348](#) for details.

6. After system generate, check the following clock information:

- View the log file, and check that there are no inferred clocks.
  - Check for derived clocks in the Clock Relationships section. Set `create_generated_clock` constraints for them in the FDC.
  - Run SLTA and check results. See [Running System-Level Timing Analysis \(SLTA\), on page 547](#) for details.
7. Run the following checks at the map stage:
- After mapping, check the Clock Relationships section to make sure that asynchronous domains have been properly defined for asynchronous clocks. If needed, adjust constraints and rerun mapping.

## Running System-Level Timing Analysis (SLTA)

System level timing analysis (SLTA) gives you timing estimates of the inter-FPGA paths and this can help identify bottlenecks. It includes timing models for TDM, cable, and I/O delays that local static timing analysis does not account for. You can run SLTA to analyze all the data paths between FPGAs after partitioning, instead of analyzing individual path segments. You can also run SLTA with backannotated results from place and route.

1. Partition the design as usual (run `pre_partition`, `run partition`, `run system_generate`).
2. To generate a report using backannotated place-and-route data, do this:
  - Synthesize, place, and route the FPGAs. You can run the jobs in parallel using the `launch protocompiler -script` command.
  - Import the place-and-route data to backannotate the design, using the database `apply_state -import_vivado` and database `apply_state -backannotate` commands. See [Importing Place and Route Results for Backannotation, on page 584](#) for details.
3. From the `system generate` database state, run SLTA with the `report timing -generate` command.

The `report timing -generate` command uses the mapped databases for the FPGAs to generate a system-level report with timing estimates for the inter-FPGA paths in your design. It considers cable length, bit rates, TDM ratios and I/O delays. The complete syntax for the command is described in [report timing, on page 127](#) in the *Command Reference Manual*.

Use the `-mode` argument to select the netlist to analyze and determine the kind of report generated. To see a list of valid netlists, use the `report timing -list` command. The more mature your design, the better the accuracy of the timing estimates. The following table shows what is reported at different stages of the design.

| <b>System Generate Stage</b>                                                                                              | <b>Reports ...</b>                                                                                           |
|---------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| Partitioned design, after TDM insertion<br>report timing -generate -mode                                                  | Inter-FPGA paths with delay estimates.<br>The TDM circuit is modeled as a lump delay.                        |
| Partitioned design, after TDM insertion, with synthesized, placed, and routed FPGAs<br>report timing -generate -mode slta | Inter-FPGA paths with backannotated delays from place and route. The TDM circuit is modeled as a lump delay. |

The SLTA timing report covers inter-FPGA paths. HSTDPM is treated as through points.

There are some limitations when you use backannotated results to run SLTA:

- The accuracy of SLTA depends on the backannotation success rate.
- Only net delays are backannotated, not cell delays.
- SLICE packing might prevent some nets from being updated.
- TDM is an approximation, and not the actual physical result.
- Do not modify your top-level timing constraints, as this can lead to unpredictable results.

4. Check the results and make adjustments to the design.
  - Check if the selected HSTDPM ratios meet performance targets.
  - Check for clock skews between FPGAs, especially if clocks cross on HT3 connectors.
  - Check for other performance bottlenecks like multi-hop paths.
  - For SLTA with backannotation, check inter-FPGA data path delays and clock path delays. Verify backannotated data by checking the location information in the report.

If both the Vivado timing report for the intra-FPGA paths and the back-annotated SLTA report are clean, it is a good indicator that the prototype will run on the hardware at the specified frequency.

# Handling Errors and Warnings

See the following topics for more information:

- [Working with Errors and Warnings](#), on page 550
- [Manipulating Message Display and Reporting](#), on page 551
- [Working with Downgradable Errors and Other Message Categories](#), on page 555

## Working with Errors and Warnings

Open the log file or report as described in [Checking Reports and Log Files](#), on page 512 and analyze the messages. Errors are prefixed with @E and warnings are prefixed with @W. You must resolve all errors and check all warnings.

1. Resolve all errors.

- Search for messages with an @E (error) prefix, which indicate error messages.
- If a message references an error in a file that cannot be exported, use the edit command to open the file, as described in step 6 of [Checking Reports and Log Files](#), on page 512.
- Fix all errors, because you cannot continue with the implementation of a design with errors. If the message is not self-explanatory or if you are unsure about how to handle it, click the message ID to go to the documentation for the message.
- If the software crashed, check for error messages that help narrow down the search and pinpoint what caused the crash. For example:

`@E BN591 More debug information: The last object processed before the error was <objectName> in module <moduleName>.`

You can then check the specified object, module, or view to identify errors and fix them. Depending on the design, you could black-box the identified module and continue working on the rest of the design.

2. If you get warnings (@W prefix) after a run, do the following:

- Search for and review messages with a @W (warning) prefix.

- Check the warnings and make sure you understand them. Decide if the situation is something you need to act on, or whether you can ignore it.
  - For GUI and HTML reports:  
If the message is not self-explanatory or if you are unsure about how to handle the error, click the message ID to go to online information about the condition that generated the warning.
3. Check other messages.
    - Skim through the notes (@N). You can usually ignore notes, because they are typically informational in nature.
    - If you see Automatic dissolve at startup messages, you can usually ignore them. They indicate that the tool has optimized away hierarchy.
  4. You can bookmark messages as described below:
    - Press Ctrl-f.
    - Type the prefix you want as the criteria on the Find form (for example, @W) and click Mark All. The software inserts bookmarks at every line that has the prefix you specified. You can now page through the file from bookmark to bookmark using the commands in the Edit menu.

You can display only the messages you want to see, using the techniques described in *Manipulating Message Display and Reporting, on page 551*.

5. To save a report, use the export report command and specify a directory location.  
Do this if you want to save or manipulate results from a file, because the database files cannot be accessed directly.

## Manipulating Message Display and Reporting

You can display only the messages you want to see by suppressing them. For a certain class of messages, you can also change their designation and downgrade them. You can also upgrade other messages to errors. Overriding a message with another message designation lets you elevate or downplay a particular message. You can also set limits and specify the messages to be reported. You can do this from the GUI or through Tcl, as described below.

- [Overriding Message Designations and Suppressing Output, on page 552](#)

- [Setting Limits on the Number of Log File Messages](#), on page 554
- [Generating a Separate Error Message Report](#), on page 555
- [Working with Downgradable Errors and Other Message Categories](#), on page 555

## Overriding Message Designations and Suppressing Output

You can change certain message designations or suppress messages, using either the GUI or Tcl.

1. Create a Tcl file that contains `message_override` commands that customize the display of messages.
  - Start the file with `message_override -clear`, so that previous override settings are negated.
  - Specify `message_override` commands to customize the display of a message. For example, you can use this command to designate downgradable errors non-fatal errors (DE prefix) as warnings, or upgrade warnings to errors. See [Working with Downgradable Errors and Other Message Categories](#), on page 555 for a description of specific changes you can make to each category of messages.

| To ...                                               | Use ..                                               |
|------------------------------------------------------|------------------------------------------------------|
| Treat a DE message as a warning                      | <code>message_override -warning messageIdList</code> |
| Treat a warning or critical warning (CW) as an error | <code>message_override -error messageIdList</code>   |
| Treat a warning as a note                            | <code>message_override -note messageIdList</code>    |

Do not include the @ symbol when you specify the message number. For example:

**`message_override -warning DE130`**

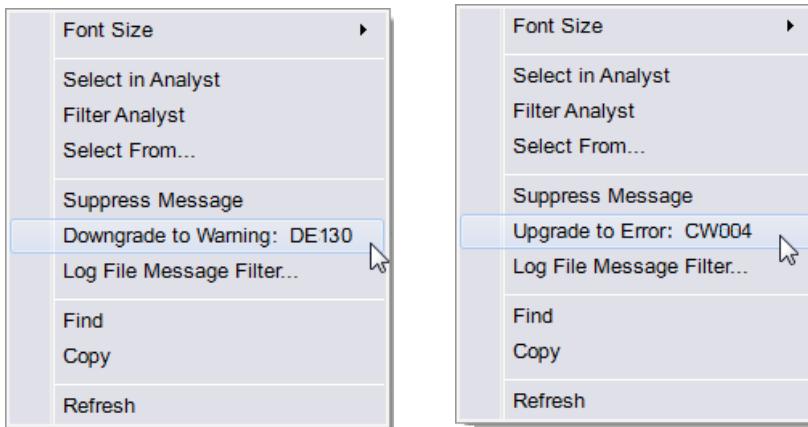
- Specify limits on the number of messages displayed with the `-suppress`, `-limit` and `-count` arguments to `message_override`. See [Setting Limits on the Number of Log File Messages](#), on page 554 for details.
- Source the Tcl file from the tool window when you want to change a setting, or when you first start the tool.

To automatically invoke the message settings every time you start the tool, add the source command to any of the setup files listed below. These are the setup files for the installation, personal setup, and the current working directory, respectively, and the tool reads them in that order.

```
installDir.synopsys_pc_setup
~/synopsys_pc_setup
./synopsys_pc_setup
```

Instead of creating a Tcl file, you can alternatively type individual message\_override commands in the tool window.

2. Do the following to manipulate message output from the GUI:
  - Go the message ID in the log file.
  - Right-click the five-character message code, and select the appropriate command to upgrade or downgrade the message from the popup menu. You can only downgrade errors with a DE message prefix, and only upgrade warnings with a CW prefix.



The selected action moves the message from the upper to the lower section of the message filter. The Override column reflects the disposition of the message. The TCL window prints a message to confirm the change. For example:

```
message_override -warning DE130
```

3. To see the changes reflected, rerun the state in database state where you changed the message designation.

For example if you suppressed a message in the c0 compile database state, recompile the c0 state to see the changes.

The tool handles the changes as shown in the following table:

| ID Status Change     | Tool Action                                                                                                                       |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| Changed to an error  | Stops executing at the first occurrence of the message and prints the message in the @E: <i>msgID</i> : <i>messageText</i> format |
| Changed to a warning | Prints the message in the @W: <i>msgID</i> : <i>messageText</i> format.                                                           |
| Demoted to a note    | Prints the message in the @N: <i>msgID</i> : <i>messageText</i> format.                                                           |
| ID suppressed        | Excludes the message from the srr file.                                                                                           |

Modifying the classification does not affect the message string for reporting. A message originally categorized as an error continues to be reported as an error regardless of its user-assigned status.

4. Change the message classification back to its original designation.
  - Fix the cause of the upgraded warning (critical warning) or complete the rest of the flow (downgradable error).
  - To use Tcl to change a message back to its original designation, use the `message_override` command with the original designation argument. For example:

**`message_override -error DE130`**

- To use the GUI to change back to the original classification, right-click one of the modified messages in a report and select the appropriate Revert command from the pop-up menu.
- Rerun the state again to confirm that the message status has reverted to the original.

## Setting Limits on the Number of Log File Messages

The number of messages reported in the log file can become overwhelming. Use the `-suppress`, `-count` and `-limit` arguments to the `message_override` command to control the number of messages reported.

1. Use -suppress to prevent specified notes and warnings from being reported. You cannot suppress errors.

```
message_override-suppress BN10
```

2. Use -limit to set the number of messages reported in the log file.

This example limits MF580 and MF581 to 1000 each in each log file:

```
message_override-limit {MF580 MFS581} 1000
```

3. To report all occurrences of a particular message, use -limit and -count unlimited together:

```
message_override -limit {CL118} -count unlimited
```

## Generating a Separate Error Message Report

Use the report message command to create custom message reports; for example you can generate reports that contain just the @E messages.

1. To generate a list of messages from a specific report file, use the report message *reportName* command.
2. To generate a list of all messages, use the report message command with no arguments.

## Working with Downgradable Errors and Other Message Categories

Downgradable errors are specially designated messages whose classification can be temporarily changed to warnings. They have a special DE prefix that identifies them.

You can downgrade or upgrade messages from the GUI or through Tcl, as described in [Overriding Message Designations and Suppressing Output, on page 552](#).

The following table describes the message designations you can change:

|                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Downgradeable errors<br>(@DE) | Can be downgraded to warnings. This is a small set of <i>non-fatal</i> errors where you can temporarily postpone addressing the error and continue with the design flow and verification of other aspects of the design.<br><br>For example if you are pipecleaning a design, you can downgrade errors about tight user clock constraints or incomplete trace assignments, even though this negatively affects the results, because the quality of the results is not your focus at that point. You must deal with the overconstraining or assignment errors on later runs. See <a href="#">Overriding Message Designations and Suppressing Output , on page 552</a> for comprehensive details. |
| Errors<br>(@E)                | Cannot be downgraded or suppressed.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| Critical warnings<br>(@CW)    | Can be upgraded to errors or suppressed. This small set of warnings represent critical problems. Elevating them to errors ensures that they are recognized and dealt with, because the error status forces the tool to stop.                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Warnings<br>(@W)              | Can be upgraded to errors or suppressed. Elevating them to error status forces the tool to stop when it encounters them.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| Notes<br>(@N)                 | Can be upgraded to warnings or suppressed.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |

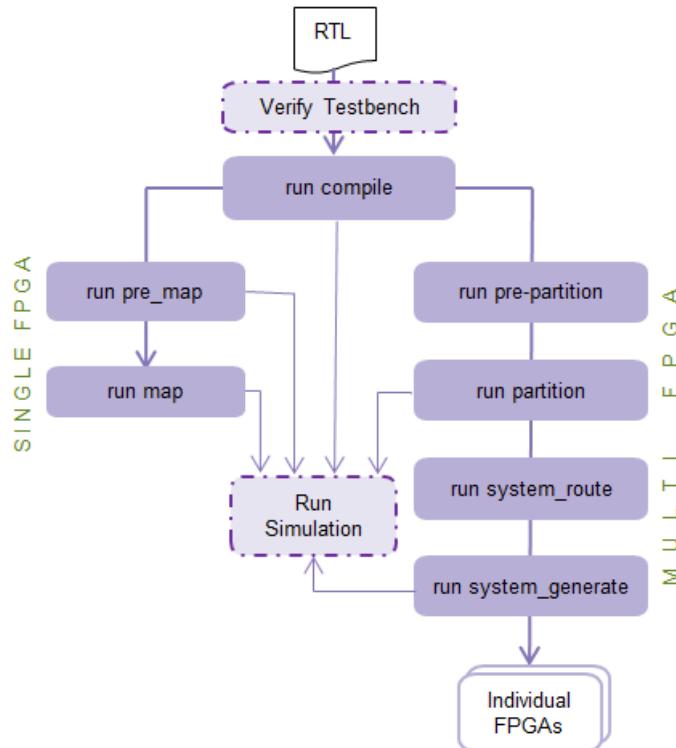
# Running Simulation

You can check generated RTL by running simulation with the Synopsys VCS® product, a high-capacity simulator for Verilog code. The first step is to verify the testbench. After that, you follow the normal steps in the design flow and simulate at different points to check your design against the RTL.

You can run VCS simulation on native RTL or after mapping. VCS is the recommended tool for simulation because it includes functionality to handle HAPS-specific encrypted RTL, and because the PLI server for UMRBus is pre-compiled in the VCS tool.

## Simulation Flows

You can run simulation on a single-FPGA or multi-FPGA design.



See the following for details about the steps in the flow:

- [Setting up the Tools and Testbench](#), on page 558
- [Running Gate-Level Simulation for a Single-FPGA Design](#), on page 560
- [Running RTL Simulation for a Multi-FPGA Design](#), on page 562
- [Simulating Post-Partition Non-TDM Designs](#), on page 567
- [Simulating Post-Partition TDM Designs](#), on page 568
- [Running Gate-Level Simulation for a Multi-FPGA Design](#), on page 573
- [Generating a Testbench from FSDB](#), on page 575
- [Running Place and Route](#), on page 576

## Setting up the Tools and Testbench

The following steps describe how to set up the tools and verify the testbench so that you can run simulation with VCS.

### 1. Set up the tools.

- Make sure the VCS, Confpro, and Vivado tools are installed. Vivado is required to run gate-level simulation, because it includes libraries for Xilinx primitives. Confpro is required for simulation with multi-FPGA designs, because it includes the HAPS supervisor model.
- Set up the environment variables for the tools needed to run simulation:

```
export VCS_HOME=path
export XILINX_VIVADO=path
export CONFPRO_PATH=path
export VM_NETLIST_DIR =path
```

|                |                                                                                                                                                                |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| VCS_HOME       | Path to the VCS installation; required.                                                                                                                        |
| XILINX_VIVADO  | Path to the Vivado installation; required for Xilinx primitive libraries.                                                                                      |
| CONFPRO_PATH   | Path to the Confpro system software; required for HAPS supervisor models. (Multi-FPGA designs only.)                                                           |
| VM_NETLIST_DIR | Path to the directory where synthesis files will be copied for simulation; required for top-level simulation in multi-FPGA designs. (Multi-FPGA designs only.) |

When you set VM\_NETLIST\_DIR, the tool replaces placeholders in the top-level wrapper file (top\_hdl.vp) with the directory you specified in VM\_NETLIST\_DIR when you run System Generate. When you export netlists, the synthesis files in this directory include a partitioned netlist for each FPGA (fb1\_uA.vm, fb1\_uB.vm...) as well as a *design.hapsinit.vm* file for each FPGA (fb1\_uA\_hapsinit.vm, fb1\_uB\_hapsinit.vm...) Specifying the directory with the files ensures that VCS finds the .vm netlists and *design.hapsinit* files for the individual partitions when simulation is run.

2. Make sure you have the following additional files to run simulation:
  - Original RTL source files for comparison
  - VCS testbench
  - For gate-level simulation, add the Xilinx primitives library:  
\$XILINX/verilog/src/unisims/\* To simulate the GPIO interface, make sure to include the appropriate GPIO .v file in the simulation: For example:

```
installDir/lib/synip/gpio/verilog/HAPS70_2000T_GPIO.v
installDir/lib/synip/gpio/verilog/HAPS70_GPIO.v
```

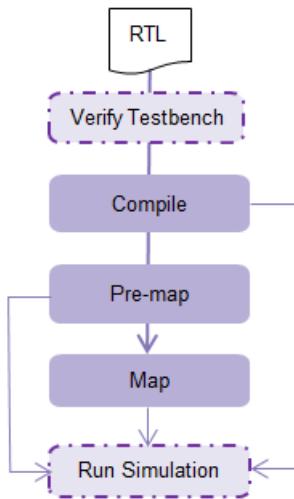
3. Verify the testbench by simulating it.

It is recommended that you use VCS to validate the testbench with the original RTL, before running the prototyping tool. Once it is verified, you can use the testbench to run simulation.

You can now run simulation. You can run gate-level or RTL simulation on your design, depending on the design stage. See the following for more information:

- [Running Gate-Level Simulation for a Single-FPGA Design](#), on page 560
- [Running RTL Simulation for a Multi-FPGA Design](#), on page 562
- [Simulating Post-Partition Non-TDM Designs](#), on page 567
- [Simulating Post-Partition TDM Designs](#), on page 568
- [Running Gate-Level Simulation for a Multi-FPGA Design](#), on page 573

## Running Gate-Level Simulation for a Single-FPGA Design



The following steps show you how to run gate-level simulation for a single FPGA and functionally verify the RTL. You can run VCS simulation after the compile, pre-map, or map stages.

1. Complete the setup and validate the testbench.

See [Setting up the Tools and Testbench, on page 558](#).

2. Load the database you want to simulate.

You can run simulation on compiled, pre-map, or mapped databases.

For example:

```
database set_state {m0}
```

For the testbench, it is recommended that you initialize all inputs to the design to a known logic value; avoid z or x initial values. For example:

```
reg in1;
reg in2;
...
initial
begin
#0;
in1 <= 1'b0;
in2 <= 1'b0;
...
```

When you synthesize the design, the tool generates the compiled or synthesized netlist (.vm). Different stages generate different files for simulation:

|                   |                                                                                       |
|-------------------|---------------------------------------------------------------------------------------|
| Compiled database | <i>design_compile.vm</i> (Netlist)                                                    |
| Pre-map database  |                                                                                       |
| Mapped database   | <i>design_compile.vm</i> (Netlist)<br><i>design_hapsinit.vm</i> (HAPS Initialization) |
|                   |                                                                                       |

3. Export the files needed for simulation using the `export netlist` command.

```
export netlist -path directoryPath
```

This command writes out the vm netlist file to the specified location. If run on a mapped design, it also writes out the *design\_hapsinit.vm* file, which is only generated after mapping. This file is required to correctly handle the mapped design in simulation.

4. If running simulation on a mapped design, instantiate the `hapsinit.sim` module in the testbench.

Make sure to connect `clk_0` to the inverted clock net. For example: // Required testbench changes:

```
dut dut_inst (
.clk (clk_net),
.reset (reset),
.clk_0 (~clk_net),);
```

```
hapsinit_sim hapsinit_sim_uut ();
```

This module is only generated after mapping. It is not required for simulation at earlier stages of the design.

5. Launch the VCS software and run simulation.

- This first step is optional. If you have a mixed language design or need to analyze libraries, use the vhdlan or vlogan commands to run analysis.
- If you are simulating a mapped design, make sure to specify the *design\_hapsinit.vm* file as input to the vcs command. Specify it after the model file. For example:

```
vcs design.vm hapsinit.vm testbench libraries
```

You do not need this file to run simulation after compile.

- If you have Xilinx primitives, specify the Xilinx simulation libraries from the Vivado installation (`$XILINX_VIVADO/data/verilog/src/unisims`).
- Run simulation with the VCS simv command.

## Running RTL Simulation for a Multi-FPGA Design

You can use VCS to run RTL and gate-level simulation at different points in the design cycle, as shown below. A HAPS system contains one or more supervisors called CDE and one or more FPGAs. From the System Generate stage, you can run RTL simulation after generating the design partitions, or you can run gate-level simulation after mapping the FPGAs.

### Overview of RTL Simulation in a Multi-FPGA Design

The following steps provide a high-level view of how to simulate a multi-FPGA design. Subsequent procedures provide details of running RTL simulation for TDM and non-TDM designs.

1. Generate the results you want to simulate.
  - Set up variables and files for simulation ([Setting up the Tools and Testbench, on page 558](#)).
  - Make sure you have Confpro set up.
  - Run the appropriate commands in the normal design flow sequence to partition the design, through `run system_generate`.
2. From the System Generate stage, export the files required for simulation, using the appropriate export command:

| To Run ...                                                                | Command                                                                                           |
|---------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| System-level post-partition simulation (no TDM)                           | export netlist<br>See <i>Simulating Post-Partition Non-TDM Designs</i> , on page 567.             |
| System-level post-partition simulation (with TDM)                         | export netlist<br>See <i>Simulating Post-Partition TDM Designs</i> , on page 568                  |
| Gate-level simulation on the entire mapped system (System Generate state) | export netlist<br>See <i>Running Gate-Level Simulation for a Multi-FPGA Design</i> , on page 573. |

The `export netlist` command generates the required files and a `.do` script to run simulation in a `synvcs` directory. See *synvcs Directory for VCS*, on page 563 for more information about the files it includes. After mapping, you additionally require the `export netlist` command, to generate the mapped netlists and other required files.

3. Edit the testbench and script as needed.
4. Run simulation.

#### synvcs Directory for VCS

This directory is generated by the `export netlist` command when run from any stage in a multi-FPGA design. It contains the files needed to run VCS simulation. The `synvcs` directory is at the same level as the database. If you run simulation again or at a subsequent design stage, it overwrites the contents of this directory.

- **VCS .do script**

This is a master script that contains pointers to all the required files needed to run simulation for a multi-FPGA design; for example, information about VCS and Xilinx setup, the RTL files, machine-generated files, CAPIM interfaces, XMRs, and so on. The name varies slightly, according to the design stage from where it was generated. This is a template that automatically includes the information or placeholders for the information that you would otherwise have to enter manually. You must edit it to match your setup and the design state. See *Example .do File*, on page 565 for an example.

- Testbench

Verify the testbench file before running simulation. You might have to edit this file, depending on the kind of simulation you are running.

- System files

These files model the entire system, and are required even if you are simulating a single-FPGA system.

File Required for Both RTL and Gate-Level Simulation

|                                                                                              |                            |
|----------------------------------------------------------------------------------------------|----------------------------|
| <code> \${CONFPRO_PATH}/data/haps/<br/> hapsSystem/sim/<br/> CDEhapsModelNumber_sim.v</code> | HAPS supervisor model file |
|----------------------------------------------------------------------------------------------|----------------------------|

File Required for RTL Simulation Only

|                        |                                                                                                                     |
|------------------------|---------------------------------------------------------------------------------------------------------------------|
| <code>design.vp</code> | Board file that defines the top level of the partitioned design, and contains instantiations of the different FPGAs |
|------------------------|---------------------------------------------------------------------------------------------------------------------|

- User design files

These are the required files that describe the design:

Files Required for Post-Partition RTL Simulation

|                                                                              |                                                                                                                    |
|------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|
| FPGA .v files                                                                | Defines the top level for each FPGA                                                                                |
| Common module files:<br><code>./synthesis_files/common_files/module.v</code> | Defines modified modules that were partitioned into different FPGAs. Include all design files from this directory. |
| Original RTL .v files                                                        | Defines modules that were not modified.                                                                            |

Files Required for Gate-Level Simulation

|                |                                  |
|----------------|----------------------------------|
| FPGA .vm files | Gate-level netlist for each FPGA |
|----------------|----------------------------------|

- Library design files

These library files are required for vendor encrypted modules or primitives, and for the CDE module used for TDM:

### File Required for RTL Simulation Only

|                                                          |                                                                                                                           |
|----------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| <code>./synthesis_files/common_files/tdm_models.v</code> | Defines the HSTDMD simulation models and fast clock for TDM. See <a href="#">HSTDMD and TDM Simulation</a> , on page 571. |
|----------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|

### Files Required for Both RTL and Gate-Level Simulation

|                                             |                                    |
|---------------------------------------------|------------------------------------|
| <code>\$XILINX/verilog/src/unisims/*</code> | Xilinx vendor simulation libraries |
|---------------------------------------------|------------------------------------|

|                   |                                   |
|-------------------|-----------------------------------|
| <code>path</code> | Simulation libraries for other IP |
|-------------------|-----------------------------------|

- Library mapping file

The `synopsys_sim.setup` file contains VCS library mapping information.

## Example .do File

```
! / bin / ksh

set various environment variables
VCS_HOME <VCS path>
BUILD <BUILD path>
SYNOPSYS <SYNOPSYS path>
LIB <BUILD LIB path>
CONFPRO_PATH <CONFPRO path>
XILINX <XILINX path>
XILINX_VIVADO <XILINX_VIVADO path>
VM_NETLIST_DIR <VM netlist directory path>
mkdir ./work
mkdir ./work
#$VCS_HOME/bin/vlogan -full164 -work work -file
filelist_vlog_work.txt
+incdir+$LIB/board+/myDesignDir+$BUILD/lib/board/haps+$SYNOPSYS/dw
/sim_ver -y $SYNOPSYS/dw/sim_ver +libext+.v

Analyze xilinx glbl.v file
$VCS_HOME/bin/vlogan -full164 +v2k $XILINX/verilog/src/glbl.v

$VCS_HOME/bin/vlogan -full164 +v2k
$XILINX_VIVADO/data/verilog/src/glbl.v

Analyze hypermods.v file
$VCS_HOME/bin/vlogan -full164 -sverilog $BUILD/lib/vlog/hypermods.v
```

```
Analyze board file
$VCS_HOME/bin/vlogan -full164 +v2k
$BUILD/lib/board/haps/daughterboards.v

Analyze supervisor and umr interface modules
$VCS_HOME/bin/vlogan -full164 +v2k
${CONFPRO_PATH}/data/haps/haps70/sim/CDE70_sim.v
-timescale=100ps/100ps +define+__UMRPLI_NO_SIM

$VCS_HOME/bin/vlogan -full164 +v2k
${CONFPRO_PATH}/data/haps/haps80/sim/CDE80_sim.v
-timescale=100ps/100ps +define+__UMRPLI_NO_SIM

Analyze umr capim file
$VCS_HOME/bin/vlogan -full164 +v2k $BUILD/lib/vlog/umr_capim.v
$VCS_HOME/bin/vlogan -full164 +v2k $BUILD/lib/haps/haps_simlib.v

Analyze technology files
$VCS_HOME/bin/vlogan -full164 +v2k -y $XILINX/verilog/src/unisims/
+libext+.v $XILINX/verilog/src/unisims/*.v

$VCS_HOME/bin/vlogan -full164 +v2k -y
$XILINX_VIVADO/data/verilog/src/unisims/ +libext+.v
$XILINX_VIVADO/data/verilog/src/unisims/*.v

$VCS_HOME/bin/vlogan -full164 -nc -sverilog +v2k -f
$XILINX_VIVADO/data/secureip/secureip_cell.list.f

Analyze top vp file
$VCS_HOME/bin/vlogan -full164 -work work
/myDesignDir/synthesis_files/top_hdl.vp +define+postsynth_sim
+v2k
+incdir+$LIB/board+/myDesignDir+$BUILD/lib/board/haps+$SYNOPSYS/dw
/sim_ver -y $SYNOPSYS/dw/sim_ver +libext+.v

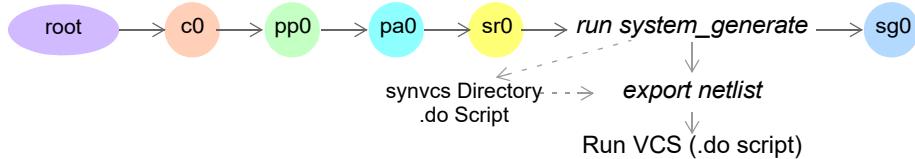
Assuming user test bench is tb.vt and is located at the following
directory
#$VCS_HOME/bin/vlogan -full164 +v2k -work work
/myDesignDir/synthesis_files/tb.vt
+incdir+/myDesignDir/synthesis_files

Do Elaboration
#$VCS_HOME/bin/vcs -full164 -debug_all -lib work work.tb work.glbl
-o simv_slp -R
```

```
#if [! -f ./simv_slp]
#then
 #echo "Error (sim_slp): Elaboration did not create ./simv_slp.
No simulation is possible."
 #rm -rf ./work
#exit 1
#fi
./simv_slp > sim_slp.log
```

## Simulating Post-Partition Designs

Run post-partition simulation at the system level, to validate the multi-FPGA design and ensure that the partitioned design is functionally the same as the original RTL. Run it after the system generate phase on designs with (TDM) or without (non-TDM) multiplexed traces between systems.



See the following for details about the simulation process illustrated above:

- [Simulating Post-Partition Non-TDM Designs](#), on page 567
- [Simulating Post-Partition TDM Designs](#), on page 568
- [Editing the Post-Partition Simulation .do Script](#), on page 570
- [HSTDMD and TDM Simulation](#), on page 571

## Simulating Post-Partition Non-TDM Designs

The following steps show you how to run simulation on a post-partition multi-FPGA design without TDM, to ensure that the partitioned design is functionally the same as the original RTL. See the figure in [Simulation Flows](#), on page 557 for a view of the design flow.

1. Set up the design and validate the testbench.

See [Setting up the Tools and Testbench](#), on page 558.

2. Follow the normal design flow, and partition the design with the run system\_generate command to implement the partitions.

Do not use unified compile, because the references with this flow are to VCS decompiled files rather than the original files. If you are using unified compile, simulate after compile.

3. Export the files needed for simulation using the export netlist command:

```
export netlist -path dirPath
```

Ignore messages about enabling verification mode; this is not required for simulation.

The tool also generates a VCS sim\_slp\_post\_synth\_script.do script and testbench, and exports all required files to a VCS directory. See [synvcs Directory for VCS, on page 563](#) for a description of some of the important files used for VCS simulation.

4. Edit the sim\_slp\_script.do script.

See [Editing the Post-Synthesis Simulation .do Script, on page 574](#) for details.

5. Run the .do script to simulate the design with VCS.

## Simulating Post-Partition TDM Designs

The following steps show you how to run simulation at the system level on a multi-FPGA design after partitioning, to ensure that the design is functionally the same as the original RTL. See the figure in [Simulation Flows, on page 557](#) for a view of the design flow.

1. Set up the design and validate the testbench.
  - See [Setting up the Tools and Testbench, on page 558](#) for setup information.
  - Make sure that the design follows the guidelines described in [HSTDm and TDM Simulation, on page 571](#).
2. Partition the design.
  - Follow the normal design flow for partitioning. However, do not use unified compile, because the references with this flow are to VCS decompiled files rather than the original RTL. If you are using unified compile, simulate after compile.

- Assign nets for the TDM scheme you are using: HSTDMD or ACPM.
  - Use the `run system_generate` command to implement the partitions with the selected multiplexing scheme.
3. Export the files needed for post-partition simulation, using the `export netlist` command.

```
export netlist -path dirPath
```

Ignore messages about enabling verification mode; this is not required for simulation.

The tool also generates a VCS `sim_slp_script.do` script and testbench, and exports all required files to a VCS directory that is located at the same level as the design database. The directory also contains the generated `do` script and the design testbench. See [synvcs Directory for VCS, on page 563](#) for a description of some of the important files used for VCS simulation.

4. Edit the `sim_slp_script.do` script so that it matches your setup.

See [Editing the Post-Partition Simulation .do Script, on page 570](#).

5. For HSTDMD designs, add force statements in the testbench for each FPGA that contains HSTDMD:

```
force FPGA_instance.hstdm_clkgeninst.hstdm_clkgeninst_SIMULATION = 1
```

Use the force statements to reduce runtime taken by HSTDMD training. See [HSTDMD Simulation Factors, on page 571](#) for an explanation and an example.

6. Specify the TDM models file as a library.

See [TDM Simulation Factors, on page 572](#) for details.

7. Run the edited `.do` script to simulate the design.

## Minimizing Partition Design Changes for Easier Simulation

When a design is partitioned, the tool might implement logic differently or make hierarchy changes based on the partitions. The changes make it harder to make one-to-one comparisons in simulation. The following guidelines list some things you can do to minimize design changes, avoid simulation mismatches, and make post-partition simulation go more smoothly.

- You get a VCS error if the original top-level module file (.v) and the .vp file with the partitioned top-level information have the same name. If the top-level file does not have any other module definitions, remove it from the filelist to avoid this error.
- Manually add the NGC files.
- If you have VHDL complex datatypes, parameters might be dropped. Check the generated partition files and edit them manually as needed.
- If you have partitioned Verilog modules with configuration defined at the top level, the configuration might not be applied after partitioning. Check, and edit the configuration file as needed.
- Check for modules with the same name compiled in different libraries. These might not be linked properly and can cause errors.
- You cannot simulate if you used unified compile flow because the netlists for the partitioned FPGAs refer to VCS decompiled files rather than the original files.

## **Editing the Post-Partition Simulation .do Script**

Before running post-partition simulation, edit the sim\_slp\_post\_synth\_script.do script. The script automates the simulation setup, but you must edit it so that it accurately reflects your actual setup.

1. Use a text editor to open the sim\_slp\_script.do script from the synvc directory.
2. Set the environment variables mentioned in the script according to the shell being used.
3. Check the include paths.
  - If you did not use the \_\_SYN\_COMPATIBLE\_INCLUDEPATH\_\_ macro when you compiled the design, check for and manually add any missing include paths for VCS. Paths might be missing because VCS handles include paths differently. The macro eliminates the incompatibility by handling the include paths in the same way as VCS.
  - If you have a lot of include paths and library paths, the VCS command line might become too long for VCS to handle, because these paths have to be specified on the VCS command line, unlike the

RTL files. If this is the case, put each command line in a file and pass it separately to VCS.

4. If you have encrypted transactors, do the following:

- Add the simulation models to the .do file. For example, add Xactor\_sim.v, and Xactor\_simdefines.v.
- Add comments to the encrypted files, Xactor.v, Xactor\_pkg.v, and Xactor\_connect.v.

If the transactors are not encrypted, the .do file automatically passes the source files to VCS, and you do not have to do anything.

5. Replace tb.vt with your testbench name and location and uncomment the line. For example:

```
Assuming your test bench is myTestbench.vt, in the design home directory
$VCS_HOME/bin/vlogan +v2k -work work <your_path>/myTestbench.vt
+incdir+/<your_path>
```

6. Uncomment the elaboration line, and replace work.tb with your testbench module name. For example:

```
Do Elaboration.
$VCS_HOME/bin/vcs -debug_all -lib work myWork.testbench work.glbl -o simv_slp
-R
```

7. Save the script.

## HSTDm and TDM Simulation

The following topics describe pin multiplexing criteria that affect simulation.

### HSTDm Simulation Factors

In addition to the factors listed below, also refer to the other general factors listed under [TDM Simulation Factors, on page 572](#). The following factors specifically affect HSTDm simulation:

- Simulation modules and tdm\_modules.v

The tool generates HSTDm black boxes after it creates partitions with run system\_generate, which you use to run gate-level HSTDm simulation. The black boxes contain built-in simulation modules, and are defined in tdm\_models.v. This file also includes the fast clock input definition

required for TDM. You must specify this file as a library file when you run simulation.

- HSTDMD training

HSTDMD simulation does not start until after HSTDMD training is complete, and the training process could take hours. To reduce the time taken to simulate HSTDMD training to a few minutes, enable simulation mode by forcing the following values in your testbench:

```
force FPGA_instance.hstdm_clkgeninst.hstdm_clkgeninst_SIMULATION = 1
```

Do this for all FPGAs that contain HSTDMD. The following Verilog code example illustrates how to enable simulation mode with a board instance named DUT that contains two FPGAs:

```
initial begin
#0
 force test_bench.DUT.inst_fb_uA.hstdm_clkgeninst.hstdm_clkgeninst_
 _SIMULATION = 1;
 force test_bench.DUT.inst_fb_uB.hstdm_clkgeninst.hstdm_clkgeninst_
 _SIMULATION = 1;
end
```

## TDM Simulation Factors

The following factors affect TDM simulation in general:

- *tdm\_models.v*

The tool generates this file after run system\_generate, after it creates partitions. It contains the fast clock input definition required for TDM. You must specify this file as a library file when you run simulation.

- Clock relationships

Post-partition RTL simulation automatically inserts a fast clock generator for simulation modules. If there is a flip-flop-to-flip-flop path where both flip-flops are clocked by the same clock, the clock relationship between user clocks and the fast clock must meet the following constraint:

$$\text{User clock period} > (\text{ratio} + 1) * \text{fast clock period}$$

The default fast clock period defined in the HSTDMD\_SIM\_offchip\_clock\_gen module is 1 ns (1 GHz).

## Running Gate-Level Simulation for a Multi-FPGA Design

Gate-level simulation verifies a mapped database. You can run post-synthesis simulation at the system level for a multi-FPGA design. The following procedure describes the details:

1. Set up the design and validate the testbench.

See [Setting up the Tools and Testbench, on page 558](#) for setup information. In particular, make sure that the environment variables are set up.

2. Complete the design

- Partition the design as usual and implement the partitions with `run system_generate`. This command sets up databases for each partition so that they can be synthesized. It also generates the `top_hdl.vp` file.
- Compile and map the individual FPGAs, as in the normal synthesis flow. When you synthesize the design, the tool generates a synthesized netlist (`design.vm`) and a `design_hapsinit.vm` file for each FPGA. These files are required to simulate a mapped design.

3. From the System Generate state, export the files needed for simulation using the `export netlist` command:

```
export netlist -path directoryPath
```

This command writes out the files, including the `vm` netlists and `design_hapsinit.vm` files, to the specified location. You can specify the same directory as the `VM_NETLIST_DIR` variable.

4. Export the files needed for system-level post-synthesis simulation, using the `export netlist` command.

Ignore messages about enabling verification mode; this is not required for simulation.

The tool generates a VCS `sim_slp_script.do` script and testbench, and exports all required files to a VCS directory that is located at the same level as the design database. The directory also contains the generated `do` script and the design testbench. See [synvcs Directory for VCS, on page 563](#) for a description of some of the important files used for VCS simulation.

When the `VM_NETLIST_DIR` variable is defined, the tool automatically fills in the paths in the `top_hdl.vp` file with the actual paths to the simulation

files. In this example from a top\_hdl.vp file, VM\_NETLIST\_DIR was set to ./Netlists, and the file reflects the paths to the files:

```
`ifdef postsynth_sim
`uselib file=./Netlists/mbl_uA_hapsinit.vm
hapsinit_sim hapsinit_mbl_uA ();
`endif
`ifdef postsynth_sim
`uselib file=./Netlists/mbl_uB.vm
`endif
```

5. Edit the sim\_slp\_post\_synth\_script.do script so that it matches your setup.  
See [Editing the Post-Synthesis Simulation .do Script, on page 574](#).
6. Run the edited .do script to simulate the design.

## Editing the Post-Synthesis Simulation .do Script

Before running post-synthesis simulation, edit the sim\_slp\_post\_synth\_script.do script.

1. Open the sim\_slp\_post\_synth\_script.do in a text editor, and set the environment variables mentioned in the script according to the shell being used.
2. Specify the top-level module. For example:

```
Analyze top vp file
$VCS_HOME/bin/vlogan -work work <your_path>top_hdl.vp
+define+postsynth_sim +v2k
+incdir+$LIB/board<your_path>/+$LIB/board/haps+$SYNOPSYS/dw/sim_ver
-y $SYNOPSYS/dw/sim_ver +libext+.v
```

3. Edit the testbench information.
  - Replace tb.vt with your testbench name and location and uncomment the line. For example:  

```
Assuming user test bench is tb.vt and is at the design home directory
$VCS_HOME/bin/vlogan +v2k -work work <your_path>/testbench.vt
+incdir+<your_path>+<FPGA database path for synthesis>
```
4. Uncomment the elaborate command and edit the module to the name of your testbench. For example:

- ```
# Do Elaboration.  
$VCS_HOME/bin/vcs -debug_all -lib work work.testbench work.glbl -o simv_slp -R  
  
5. If your testbench simulates RTL code, uncomment and edit the lines  
that specify your RTL code. For example:  
  
$VCS_HOME/bin/vlogan +v2k -work work <your_path>/file1.v +incdir+$LIB/board+ ...  
$VCS_HOME/bin/vlogan +v2k -work work <your_path>/file2.v +incdir+$LIB/board+|...
```

Running DUT/IP Separation Simulation Flow for HAPS-100

Generating a Testbench from FSDB

If you have a fast signal database (FSDB), you can generate a testbench from it instead of writing one. Use the Verdi and VCS tools to generate a testbench from an FSDB. For details of the syntax and commands, refer to the corresponding tool documentation.

1. From the Verdi tool, extract the testbench using the `fsdbextract` command.
 - Extract the FSDB for the scope that you want, which is typically the module you want to simulate. This is an example of the syntax:

```
fsdbextract inputFsdb -s scope -o outputFsdb
```

- Convert the FSDB to VCD with the `fsdb2vcd` command:

```
fsdb2vcd inputFsdb -o outputVCD
```

2. Create a `cfg` file for the VCS `vcat` functionality to generate the testbench.

The `cfg` file has three things defined inside it: `<scope>`, `testbench`, `<top part of module definition>`. This is used as input to `vcat`. The scope here must be the same as the scope defined in the previous step.

3. In VCS, create the testbench by running `vcat` with this command:

```
vcat -vgen <cfg file created in step 2> <input VCD file>
```

Use the testbench created in the previous step to run simulation in the prototyping tool.

Running Place and Route

When you are ready to place and route the design, you have two choices to run Vivado place and route: to run multiple parallel explorations to find the best place-and-route solution, or to launch and run Vivado as usual. The former intelligently sets and experiments with different PAR options it runs in parallel to come up with the best solution, while the latter runs Vivado in normal mode. It is recommended that you use the exploration option whenever possible, unless you have limited machine resources to run the parallel place-and-route runs.

For more information about the place and route techniques and the backannotation of results after place and route, see these topics:

- [Running Place-and-Route Exploration, on page 576](#)
- [Running Place and Route without Exploration, on page 580](#)
- [Customizing Scripts to Run Vivado \(Single-FPGA Designs\), on page 583](#)
- [Importing Place and Route Results for Backannotation, on page 584](#)
- [Resynthesizing After Place and Route, on page 586](#)
- [Analyzing Congestion Issues, on page 589](#)

For information about exporting the design to run on the hardware after place and route is complete, see [Preparing to Run on the Hardware, on page 614](#).

Running Place-and-Route Exploration

This procedure describes how to automatically explore different Vivado place-and-route solutions and find the best one. This is the recommended method to place and route, but alternatively, you can launch and run Vivado in the normal way ([Running Place and Route without Exploration, on page 580](#)).

1. Make sure you are ready to run place and route:
 - Start with a mapped design that is ready for place and route.
 - If you have not already done so, ensure you have Xilinx Vivado properly installed and licensed, and that you have set the appropriate

environment variables to run the place-and-route software. See [Setting Environment Variables, on page 30](#).

- Make sure that you are operating in synthesis mode, not partition mode. If necessary, source the Tcl options file again to make sure you have the correct settings.
2. Specify the maximum number of exploration jobs that can be run in parallel.
- If you are on a Linux platform, you can enable CDPL with the option set cdpl command to distribute the jobs over multiple specified machines. If CDPL is not specified, all the jobs run on the host machine in parallel, according to the CPU resources. See [Using CDPL for Distributed Processing, on page 500](#) for information about specifying CDPL.
 - Use the option set max_parallel_par_explorer command to specify the number of PAR exploration jobs to run in parallel. When it explores place-and-route solutions, the tool uses the number you specify as a guide to the number of parallel jobs to run, based on the machine resources available. If you are using CDPL and you have enough licenses, set the value to a high number so that the tool has the flexibility to determine how many jobs to run simultaneously.

On the other hand, if you are running PAR exploration without CDPL, reduce the strain on the host machine resources by limiting the number of jobs that can run simultaneously. A lower value reduces the chances of overloading resources, but the run time might increase.

3. Optionally create a custom Tcl script to use when running exploration.
- Use the standard template as a starting point. The base template files are located in \$BUILD/lib/xilinx. For details about the Vivado commands, refer to the Xilinx documentation.
 - To avoid error messages, make sure that the script includes the RUN DESIGN and GENERATE REPORTS headers shown in the example ([PAR Exploration Custom Script Example, on page 580](#)). Otherwise you get errors.
4. Specify the run par_explorer command.

The tool examines design characteristics and launches a set of Vivado PAR jobs in parallel, using options that it has intelligently selected based on the design characteristics. For example, if the design has a high

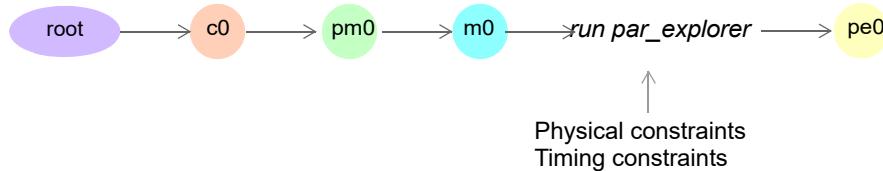
positive slack value, the command favors runtime options. The options used in the run are combinations of Vivado parameters such as area optimization, QoR placement parameters, placement parameters for congestion, physical optimization, QoR routing parameters, routing parameters for congestion, and routing parameters for runtime.

- To use a custom script, specify it with the `-script` option:

```
run par_explorer -script custom.tcl3
```

For information about using the customized script in a multi-FPGA scenario, see [Customizing Scripts to Synthesize, Place, and Route FPGAs, on page 430](#).

- If you want to place and route in the background, specify the `-bg` argument to the command. For the complete syntax, refer to [Shell Command Reference, on page 17](#) in the *Command Reference Manual*.



- To view the results from the PAR explorer database state, do this:
 - Ensure the PAR explorer state (`pe0`) is active and type `view report` in the Tcl window.
 - Check the Timing Summary Report in the Xilinx P&R Report, for an overview of the timing. Judge whether the results meet your prototype requirements.
 - Check the `area.txt` output file for area utilization summaries by slice, memory, DSP, and I/O resources of the FPGA.
 - Check the Vivado command log for a list of optimizations that were run.
- To view the entire directory for the best PAR results after exploration, run a sequence of commands like the ones shown below:

```
export vivado -path pr_1
database set {m0}
import vivado pr_1
database apply_state backannotate
```

The example above first exports the set of place-and-route results and files to an external directory (`export vivado`) and then imports them to a map database state (`import vivado`). The database `apply_state` backannotate command then backannotates the current map state with place and route information. See [Importing Place and Route Results for Backannotation](#), on page 584 for details about backannotation.

Check the results, as described in [Running Place and Route without Exploration](#), on page 580

As with other states in the process, you can always return to a previous database state and rerun it. If the results meet your requirements after place and route, you can generate a bit file as described in [Exporting Files for Runtime](#), on page 614.

PAR Exploration Custom Script Example

```
######
###      RUN DESIGN      ###
#####
#run link_design
link_design
if {[file exists "clock_groups.tcl"]} {source clock_groups.tcl}
#Evaluate options and run opt_design
eval opt_design $opt_design_flags

###      FOR INCREMENTAL FLOW      ###
puts "Flow is ${Flow}"
if (${Flow} == "Incremental") {
    #Use DCP from previous P&R run for Incremental Flow
    if {[file exists "${DesignName}.dcp"]} {
        puts "Using ${DesignName}.dcp for Incremental Place and Route"
        read_checkpoint -incremental ${DesignName}.dcp
        report_incremental_reuse
    } else {
        puts "${DesignName}.dcp does not exist. Running Place and Route"
    }
}
#Evaluate options and run place_design
eval place_design $place_design_flags
write_checkpoint -force post_place.dcp

#Evaluate options and run route_design
eval route_design $route_design_flags
#set_property BITSTREAM.General.UnconstrainedPins {Allow} [current_design]
write_checkpoint -force ${DesignName}.dcp

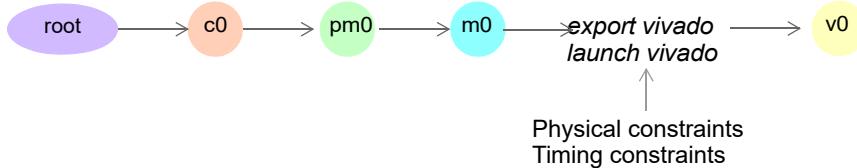
#####
###      GENERATE REPORTS      ###
#####
```

Running Place and Route without Exploration

After you have finished mapping, you can launch Vivado placement and routing as described here. However, it is recommended that you use automatic place-and-route exploration (*Running Place-and-Route Exploration, on page 576*) instead.

See *Shell Command Reference, on page 17* in the *Command Reference Manual* for details about the commands mentioned in the following procedure.

1. Start with a design that has been mapped.



2. If you have not already done so, ensure you have Xilinx Vivado properly installed and licensed, and that you have set the appropriate environment variables to run the place-and-route software.

See [Setting Environment Variables](#), on page 30.

3. Export the database with the `export vivado` command.

- You must specify the path to the target directory with `-path`. This command creates a directory with all the input files needed to run place and route. It includes the default EDIF netlist generated for Vivado, corresponding constraints in an `xdc` file, and a `run_vivado_haps.tcl` automation script, which contains design variables and commands for setup, running place and route, and reporting.

For further information about the programming interface, refer to the Vivado Design Suite documentation on Tcl commands and scripting.

- To write out the netlist for Vivado in Verilog instead of the default EDIF, include the `-format verilog` option when you specify the `export vivado` command. With this option, the command writes out a `designName.vm` file instead of EDIF. It also writes out a corresponding `xdc` file, and the `run_vivado_haps.tcl` script to run Vivado uses `vm` as the input format instead of EDIF.

If you transfer exported files to another location, you must copy all exported files. This includes encrypted files like the `.cfg` file, which contains LUT configuration information that is required for HAPS system bring-up and configuration. This file is generated for each run, and is a required file for successful bring-up.

4. Run place and route by specifying the `launch vivado` command with the name of the place-and-route Tcl script to run.

The tool launches the Xilinx place-and-route software and runs place and route using the options you set in the Tcl script. It generates the following files in the specified place-and-route directory:

| | |
|-----------------------|-------------------------------------|
| area.txt | FPGA resource utilization |
| clock_utilization.txt | Clock utilization |
| pinloc.txt | I/O assignments |
| design_post_par.xdc | I/O and logic placement constraints |
| post_route_drc.txt | DRC report |

5. To view the place-and-route results, follow these steps:
 - Ensure the place-and-route state is active and type `view report` in the Tcl window.
 - In the Xilinx P&R Report, check the Timing Summary Report section for an overview of the timing. Judge whether the results meet your prototype requirements. There are two Vivado timing summaries, one with worst case conditions and the other with optimal conditions for process variation, voltage and temperature.
 - Check the `area.txt` output file for area utilization summaries by slice, memory, DSP, and I/O resources of the FPGA.

As with other states in the process, you can return to a previous database state and rerun it. If the results meet your requirements, you can generate a bit file ([Exporting Files for Runtime, on page 614](#)).

6. To run place and route incrementally, use the design checkpoint (`dcp`) file as the starting point for the next run.

This is a typical sequence of Xilinx commands:

- Load the current design (`link_design`).
- Run `opt_design`.
- Read the incremental design checkpoint file (`read_checkpoint -incremental dcpFile`).
- Run `place_design`.
- Run `route_design`.

The tool uses an incremental flow with the `dcp` file used as a reference point. The place or route operation is run incrementally from the reference point on. Refer to the Xilinx documentation for the commands.

For ECO updates, look for commands like `disconnect_net` and `connect_net`, which are commonly used for ECO changes. Also use the Vivado schematic and search tools.

7. To use the Vivado register readback feature, see the procedure described in [*Using Global State Visibility \(GSV\), on page 753*](#).

Customizing Scripts to Run Vivado (Single-FPGA Designs)

This procedure describes how to use customized scripts to direct place and route for a standalone single-FPGA design. For information about using customized scripts in a multi-FPGA design, see [*Customizing Scripts to Synthesize, Place, and Route FPGAs, on page 430*](#).

1. Start with a mapped design.
2. Create a Tcl file with the custom options you want to run Vivado.

The options in this custom options file override the settings in the `run_vivado_haps.tcl` file. For example, you can change the default settings to generate a Verilog netlist after place and route and not write out a bit file:

```
set write_post_par_verilog "1"  
set write_bitstream_enable "0"
```

3. Specify the options file when you export the database to Vivado.

```
export vivado -vivado_option_file custom_export_Vivado.tcl
```

This example exports the options file called `custom_export_Vivado.tcl` to the top-level working directory. It also sets a variable in the `run_vivado_haps.tcl` file, so that the custom options file is sourced and the custom settings in this file are used to run Vivado. You can now run place and route as usual.

4. For customized par exploration scripts, use `export vivado` with the `-script` argument instead of `-vivado_option_file`. For example:

```
export vivado -script par_explore1.tcl
```

Using Vivado Flags and Options in Run Vivado HAPS Tcl Script

You can use Vivado flags and options in custom `run_vivado_haps.tcl` scripts during the initial exploration of a design to reduce the runtime. Most of these switches are reported by the Vivado in the log. These switches help in implementing a script-based flow without adding the switches to individual Vivado runs.

For example, you can add the Vivado switch `route.enableHoldExpnBailout` in a custom script `run_vivado_haps.tcl`:

```
catch {set_param route.enableHoldExpnBailout 1}
```

For details on the Vivado flags and options, see the Vivado documentation.

Importing Place and Route Results for Backannotation

After running place and route, you can backannotate the results from the Vivado database into a mapped state in the prototyping database.

1. Place and route the design, using the methods described in [Running Place-and-Route Exploration, on page 576](#) or [Running Place and Route without Exploration, on page 580](#).
2. Open the mapped database state from where you started place and route.
3. Specify the database `apply_state -import_vivado` command.

The command sets up the location of the place-and-route results for the mapped design state, so that you can use the place-and-route results in the mapped database state.

4. To backannotate place and route information for resynthesis or analysis in the prototyping tool, use the database `apply_state -backannotate` command.

This command imports the Vivado net delay values so that you can run more accurate timing analysis from the mapped state. It creates a new database with backannotated place-and-route information.

For information about resynthesis, see [Resynthesizing After Place and Route, on page 586](#).

5. To use backannotated information as the basis for system-level timing analysis (SLTA), do the following:
 - For each FPGA, backannotate the place-and-route results by running the database apply_state -import_vivado and -backannotate commands as described in the preceding steps.
 - From the system generate stage at the top level, run report timing -generate -mode slta command to generate a system-level timing report that is based on the backannotated information for the FPGAs. See *Running System-Level Timing Analysis (SLTA)*, on page 547 for details.

Limitation

- Does not support VM flow for backannotation.

Resynthesizing After Place and Route

If your design continues to have poor slack numbers or still fails to route after you have tried various alleviation techniques, you can run post-place-and-route resynthesis to generate a new netlist. This resynthesis run is based on information from place and route, and can improve routability and help meet timing. The run preserves most of the design and only makes changes to congested hotspot regions and timing paths with violations, so it takes only a fraction of the normal synthesis runtime.

1. Run place and route with the `export vivado` and `launch vivado` commands.

For more about placing and routing the design, see *Running Place-and-Route Exploration, on page 576* or *Running Place and Route without Exploration, on page 580*.

For post-place and route resynthesis, you can either import the results back and then resynthesize (step 2) or resynthesize without importing, based on a specified place-and-route dcp file (step 3).

2. To resynthesize based on imported information, do the following:

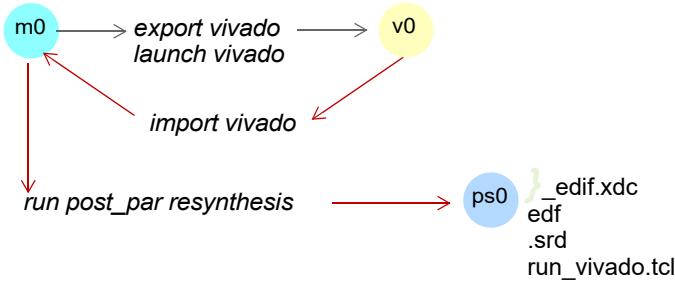
- Start with the mapped database state from where you exported the files for Vivado in step 1, and import the place-and-route database back to the same state, using the `import vivado` command.

The import operation imports the following netlist, constraint, and script files from Vivado:

| | | |
|--|-------------------------|-----------------------------|
| <code>design_edif.xdc</code> | <code>design.edf</code> | <code>run_vivado.tcl</code> |
| <code>design.dcp</code> or <code>post_place.dcp</code> | <code>design.dly</code> | |

- Resynthesize the design with the `run post_par` resynthesis command.

`run postpar_resynthesis`



You do not need to specify any arguments to the `run post_par resynthesis` command; the tool uses the imported netlist and dcp file to generate a re-mapped netlist. The command writes out new EDIF and constraint files, and a new Vivado Tcl run script. It also preserves the original SRD, EDF, and XDC files.

The command also generates a new post-place-and route state with a new netlist and constraints. The default name for the state is PS0. The PS0 state is like a mapped state, and you can look at reports, export files, or run PAR Explorer as usual. The only exception is that you cannot run Formality from this state.

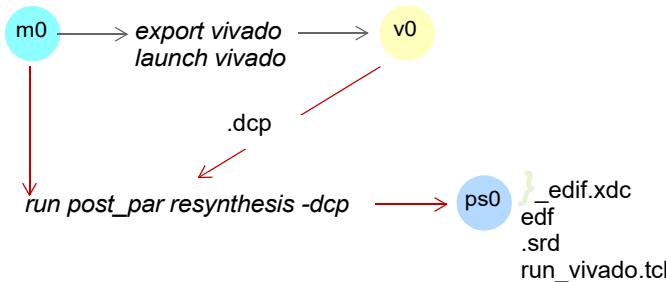
3. Resynthesize with the `run post_par resynthesis` command and the appropriate dcp file. You do not have to import the Vivado files back.
 - Start from the mapped state from which you exported files for place and route.
 - If the problem is that the design does not meet timing, specify the post-route dcp file to resynthesize the design:

```
run postpar_resynthesis -dcp path/design.dcp
```

- If the design failed to route, use the post-place dcp file for resynthesis:

```
run postpar_resynthesis -dcp path/post_place.dcp
```

The command writes out new EDIF and constraint files, and a new Vivado Tcl run script. It also preserves the original SRD, EDF, and XDC files. It also generates a new post-place and route mapped state, with the default name PS0.



4. Check that resynthesis was successful in the `pr_resyn.log` file.
5. To place and route again after resynthesizing, export the files needed to an output directory with the `export vivado -path` command.
This command exports the edif and xdc files needed for the new place and route run.
6. To run the Vivado incremental flow after resynthesis, do the following:
 - Edit the `run_vivado.tcl` script and change the flow to Incremental (set Flow “Incremental”);.
 - Rename `design.dcp` to `design_pr_resyn.dcp` to ensure that Vivado reads the dcp file.
 - Run the Vivado incremental flow.

Analyzing Congestion Issues

Congestion is design-specific. Congestion analysis is an iterative process, where you first synthesize, place, and route the design, then analyze the impact of congestion, make changes and iterate through the process again, even re-partitioning if needed.

The following sections are complementary; the first focuses on techniques to use, and the second provides specific troubleshooting guidelines for different database states:

- [Using Congestion-Reducing Techniques](#), on page 589
- [Troubleshooting Congestion at Different Design Phases](#), on page 591
- [Using TDM to Reduce SLL Congestion in Vivado](#), on page 598

Using Congestion-Reducing Techniques

Use the following techniques as you create the prototype, to minimize routing congestion at the end. It is recommended that you apply congestion techniques starting from the most recent design phases and working backwards, to minimize redesign.

1. Use the following place-and-route techniques to reduce congestion during placement and routing.
 - Spread the design logic across the FPGAs and set a 60% utilization rate for the first pass.
 - Check that the timing constraints are not too tight, as this can lead to sub-optimal placement.
 - Check the place and route strategy used. You can try different strategies in parallel with place-and-route exploration.
 - Use the congestion-specific techniques listed in the following table:

| General Techniques | Design-Specific Techniques |
|--|---|
| Congestion_BalanceSLLs place_design -directive SSI_BalanceSLLs phys_opt_design route_design -directive NoTimingRelaxation | Congestion_SpreadLogic_high place_design -directive SpreadLogic_high phys_opt_design -directive AggressiveFanoutOpt route_design -directive MoreGlobalIterations |
| Congestion_BalanceSLRs place_design -directive SSI_BalanceSLRs phys_opt_design route_design | Congestion_SpreadLogic_medium place_design -directive SpreadLogic_medium phys_opt_design -directive AggressiveFanoutOpt route_design -directive HigherDelayCost |
| Congestion_CompressSLRs place_design -directive SSI_HighUtilSLRs phys_opt_design route_design | Congestion_SpreadLogicSLLs place_design -directive SSI_SpreadSLLs phys_opt_design route_design -directive NoTimingRelaxation |

For additional information about reducing congestion at this stage, see [Congestion-Reducing Techniques at the Place-and-Route Stage, on page 592](#).

For information about running place and route, see [Running Place-and-Route Exploration, on page 576](#) or [Running Place and Route without Exploration, on page 580](#).

2. Use the design techniques listed below when you run the compile, pre-map, and map phases, to reduce congestion. Then place and route the design again and check results.
 - Run with the synthesis_strategy routability option. This option reduces the competition for routing resources by mapping LUT6 to LUT4 in high-congestion areas. This single technique might be sufficient to fix congestion issues in designs with a relatively low number of node overlaps.
 - Check for high resource utilization in the log files after partitioning and system generate, and after mapping. Aim to use less than 70% of LUT resources and less than 60% fo BRAMs and DSPs.
 - Check that logic packing is not using a high number of LUT6 primitives as compared to LUT5s, LUT4s, LUT3s or LUT2s. Make sure that the number of control sets is below 5000.
 - Avoid gated clocks on high fanout clocks.
 - Use the syn_insert_buffer attribute to insert buffers on high fanout nets.

- Do not over-constrain the design. Aggressive timing constraints with negative slacks that are 10% over the period can cause congestion.

For additional information about reducing congestion at this stage, see [*Congestion-Reducing Techniques at the Map Stage, on page 596.*](#)

3. From the system route database, check the SLL and SLR estimates in the log file:
 - SLLs are connections between SLRs. SLL usage must be below the device limit listed. If it is over the limit, try a new I/O placement or re-partition the design.
 - SLR usage must be balanced. If it is not, modify the I/O placement or re-partition the design.

For additional information about reducing congestion at this stage, see [*Congestion-Reducing Techniques at the Partitioning Stages, on page 598.*](#)

4. In a database generated after run partition or run system_generate, check the LUT utilization in the log file.

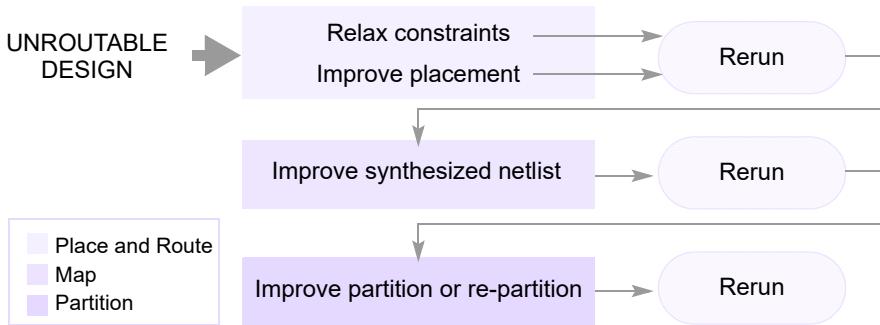
LUT utilization must not exceed 70%. If it does, try balancing the utilization by mapping RAMs to BRAMs, SRLs to registers, and arithmetic operators to DSP blocks. Alternatively, try re-partitioning.

For additional information about reducing congestion at this stage, see [*Congestion-Reducing Techniques at the Partitioning Stages, on page 598.*](#)

Troubleshooting Congestion at Different Design Phases

Congestion factors are design-dependent and you might have to use a combination of techniques to address the issues you find. Watch for typical indications throughout the design cycle, such as timing violations, high utilization numbers for LUTs, BRAMs, muxes and DSP, wide buses, a large number of interconnections between modules, many high-fanout nets, over-utilization of SLRs and long place-and-route runtimes.

Ideally, identify congestion as early as possible in the design cycle and fix the underlying issues. However, if congestion is only identified after place and route, work backwards through the flow and check whether you can address the congestion at that design stage, to minimize re-design.



The following recommendations describe techniques to try at different design stages.

- [Congestion-Reducing Techniques at the Place-and-Route Stage, on page 592](#)
- [Congestion-Reducing Techniques at the Map Stage, on page 596](#)
- [Congestion-Reducing Techniques at the Partitioning Stages, on page 598](#)

Congestion-Reducing Techniques at the Place-and-Route Stage

1. Generate the congestion map and visually check it.
 - Use the File->Open Check Point command to open the dcp file.
 - Check both horizontal and vertical congestion by right-clicking and selecting Metric->Vertical/Horizontal routing congestion per CLB.
2. Check for congestion errors.
 - Run `run_route_status` to get a full status of the routing, and then check the log file as described below.
 - Vivado error:
ERROR: [Route 35-162] 1868 signals failed to route due to routing congestion.
 - Router Utilization Summary in log file
Check the extent of the congestion (Number of Unrouted Nets and Number of Node Overlaps). Ideally, there should not be any node overlaps. The more node overlaps there are, the more congested the design.

If overall routing utilization values are not high, check for local congestion. You can also check tile areas to identify local congestion.

- vivado.log file:
Look for Verification failed messages, and CRITICAL_WARNINGS in the route_design section.

3. Check timing.

- Make sure that the paths being optimized are truly critical paths.
- Check that all timing exceptions and I/O delays are properly constrained with constraints like set_multicycle_path, set_false_path, and set_datapath_delay.
- Check routability by running place and route with relaxed constraints. Either relax the constraints in the _edf.xdc file and run place and route, or run place and route without this file. If the design then routes with this best-case scenario, re-introduce the constraints gradually and rerun. If it does not work, return to pre-partitioning and relax the system-level timing constraints. You can also try alternate placements to improve routability.

4. Check the complexity report.

- Generate an on-demand complexity report with the report_design_analysis -complexity. This report reports interconnect density per hierarchy as an exponent value.

```
report_design_analysis -complexity -hierarchical_depth numLevels  
-name complexity -file name.rpt
```

- Check that the interconnect density exponent value is less than 0.65.
- Check number of ports and number of cells per hierarchy for density.
- Check LUT, BRAM, DSP, and MUXF7 and MUXF8 usage.

5. Check SLR and SLL usage.

- Check that SLR (Super Logic Region) usage is balanced. To fix congestion caused by unbalanced SLR usage, you might have to modify I/O placement or block constraints.

If you specify advanced mode for the synthesis strategy in the prototyping tool, you can get estimates of the usage, as shown in the report below:

Estimated SLR Usage Report:

| SLR | LUT | % DFF | % BRAM | % DSP | % IO | % |
|-------|--------|-----------|--------|-------|--------|-----|
| SLR-0 | 0 | 0% 1 | 0% 0 | 0% 0 | 0% 146 | 9% |
| SLR-1 | 287831 | 94% 69030 | 11% 5 | 1% 2 | 0% 290 | 17% |
| SLR-2 | 233489 | 76% 17975 | 3% 0 | 0% 1 | 0% 326 | 19% |
| SLR-3 | 0 | 0% 0 | 0% 0 | 0% 0 | 0% 232 | 14% |

- SLL (Super Long Lines) are connections between SLR regions. Cut constraint warnings about SLLs indicate that reruns might cause routing issues. High interconnectivity and the overuse of SLLs can cause congestion. You can check for SLL usage estimates in the system generate log file for the system level, and in the post-map file for the single-FPGA level. Correct over-utilized SLLs by using a new I/O placement or by repartitioning.

If you enable the advanced synthesis option in the prototyping tool, you can get early estimates of possible problems, reported as follows:

Estimated SLL Crossing Report:

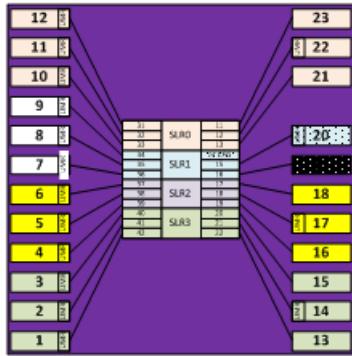
| Crossing | Available SLLs | Used | Usage |
|---------------|----------------|------|-------|
| SLR-0<->SLR-1 | 13720 | 135 | 0.01 |
| SLR-1<->SLR-2 | 13720 | 8492 | 0.62 |
| SLR-2<->SLR-3 | 13720 | 25 | 0.00 |

If multiple FPGAs are congested or if there are a large number of node overlaps, re-partition the design. If the congestion is localized on a single FPGA, try the techniques described in the next step and in [Using Congestion-Reducing Techniques, on page 589](#) first, instead of re-partitioning.

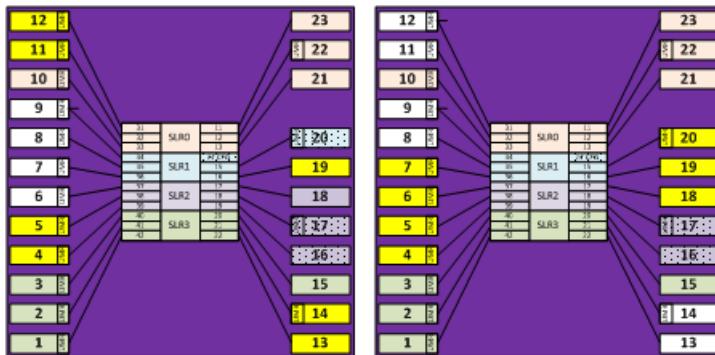
6. Redistribute connectors to balance congestion.

- If you check the congestion map and find some SLRs are under-utilized, balance connectors across SLRs. I/O placement affects connectivity because connected logic moves with the I/O. If the SLRs are highly utilized, re-partition the design.
- Fill the connectors across a single SLR, rather than vertically using multiple SLRs on just one side. The following figure illustrates this recommendation. It shows the HT3 connections on SLRs for a

HAPS-70 system, where the filled connectors in yellow (4, 5, 6, 16, 17, and 18) are all part of SLR2, and are not distributed across multiple SLRs.



- Localize the connectors instead of spreading them out, as long as this does not interfere with device utilization.
- In the following figure the example on the left with spread-out HT3 connectors (yellow) would be more congested, whereas the one on the right with a concentrated distribution of connectors would be less likely to cause congestion.



7. Try an alternative placement solution.

- Use `run par_explorer` to run different combinations of strategies automatically. Use distributed synthesis to run them in parallel.
- Select a P&R strategy based on the cause of the congestion, and run it manually. For example, run `Congestion_BalanceSLLs`. See the Vivado

documentation on implementation and design analysis for additional information on P&R strategies.

If the design remains congested, resynthesize the design to get a more routable netlist. If you have to, return to the partitioning phase and re-partition the design with congestion factors in mind. If you have to re-partition, remember that by definition re-partitioning eliminates repeatability of results, because it is a different partitioning solution. See [Using Congestion-Reducing Techniques, on page 589](#) for some techniques.

Congestion-Reducing Techniques at the Map Stage

1. Run post-P&R resynthesis.

This run uses a Xilinx dcp checkpoint file, and does not rerun all the synthesis steps. See the [Resynthesizing After Place and Route, on page 586](#) for details. Use this fast iteration for routable but congested designs, or placed designs that do not route.

2. Check timing.

- Make sure that the paths being optimized are truly critical paths.
- Check that all timing exceptions and I/O delays are properly constrained.
- Check the log file stages for constraint checker messages, high negative slack, or other timing-related warnings.

3. Check the design setup for areas that can be improved:

- Manually check that all clocks are being routed on global resources, not local ones. Use HDL Analyst to look for buffers in the mapped view. Also check that the number of BUFGs and clocks are as expected, with more BUFGs than clocks.
- Check the gated clock conversion report in the post-map log file to make sure that there are no gated clocks left unconverted.
- Check for excessive replication. If necessary, increase the global fanout threshold or use the default (10000).

4. Check for high macro usage, and balance resource utilization.

Check these attribute and directive settings and make sure that they are being used effectively and that there are no unnecessary restrictions on optimizations.

| Attribute | How to Alleviate Congestion |
|--|---|
| <code>syn_ramstyle</code> | Map RAMs to BRAMs Remove glue logic by specifying <code>no_rw_check</code> |
| <code>syn_dspstyle</code> | Map arithmetic operators to DSPs |
| <code>syn_srlstyle</code> | Map SRLs to registers |
| <code>syn_hier</code> <code>syn_noclockbuf</code> <code>syn_preserve</code> <code>syn_keep</code> | Allow free optimizations by removing these attributes when they are not needed. Use the Find in files option to search the RTL, cdc, and fdc files and remove the ones that are not needed. |

5. Turn off LUT6 packing to increase routing density.

```
option set enable_prepacking 0
```

6. For local congestion, try a different synthesis flow:

- If LUT utilization is over 60% and high performance is required, use the synthesis strategy advanced mode, which takes congestion into account when optimizing:

```
option set synthesis_strategy advanced
```

- If LUT utilization is over 60% and some timing QoR can be sacrificed for improved runtime, use the synthesis strategy routability mode, which takes congestion into account when optimizing:

```
option set synthesis_strategy routability
```

- If LUT utilization is less than 50% and high performance is not required, use the synthesis strategy fast mode, which reduces the number of optimizations:

```
option set synthesis_strategy fast
```

7. Resynthesize the design and rerun place and route.

Congestion-Reducing Techniques at the Partitioning Stages

1. In a system generate database state, make sure SLL usage is below the device limit, and that SLR usage is balanced.
If it is not, modify the I/O placement or re-partition the design.
2. In a database generated after run partition or run system_generate, make sure that LUT utilization does not exceed 70%.
If LUTs are over-utilized, try balancing the utilization by mapping RAMs to BRAMs, SRLs to registers, and arithmetic operators to DSP blocks.
Alternatively, try re-partitioning.
3. Repartition and move I/O connectors.

This is a last-resort, high-effort solution, as it involves iterating through the entire design cycle. Here are some techniques to try:

- Balance FPGA utilization by setting a bin maximum in the pcf file before partitioning:
`bin_utilization -all_bins -resource_ratio {all 0.6 }`
- Check for and reduce high TDM ratios (64 or 128) before generating partitions. High TDM ratios could prove difficult to route.
- Spread signals with high TDM ratios across connectors, because concentrating signals with high TDM ratios in the same connectors can impair design routability.
- Use this option for gated clocks when you partition the design:

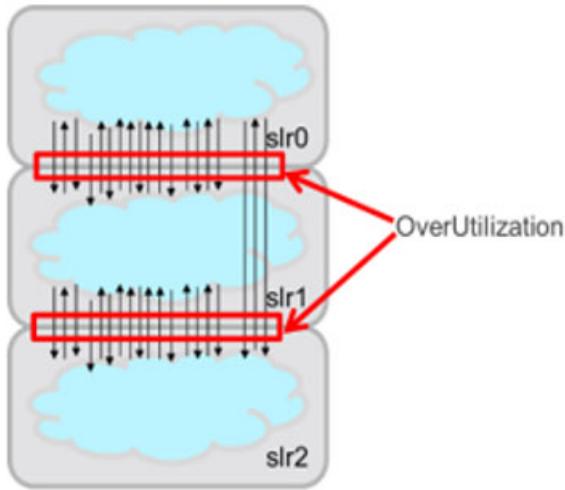
```
run partition -clock_gate_replication 1
```

Refer to [Shell Command Reference, on page 17](#) and [bin_utilization, on page 180](#) in the *Command Reference Manual* for descriptions of the run commands and the bin_utilization command, respectively.

Using TDM to Reduce SLL Congestion in Vivado

With HAPS-100 designs you can use TDM in Vivado to resolve congestion caused by high SLL demand. This lets you place and route designs that would otherwise fail with the following Vivado error message:

```
ERROR: [Place 30-99] Placer failed with error: 'SSI partitioning failed!'
```



1. If you encounter the Vivado error message about SSI partitioning failing, set the environment variable `VIVADO_SLL_TDM_ENABLE` in the ProtoCompiler tool.
2. To selectively allow or prohibit clock domain nets from being considered for TDM, set the appropriate attributes in the FDC file:

```
define_attribute {c:clkin1} {n_slltdm_clk_allow} 1  
define_attribute {c:clkin2} {n_slltdm_clk_prohibit} 1
```

Note that by default all DUT clocks are considered for SLL TDM optimization in Vivado. so use the attributes if you want to only run TDM on specific nets.

3. Run Vivado using the ProtoCompiler-generated Vivado script.

This procedure might not work with custom scripts or scripts generated with versions before R-2020.12.

4. Check for messages after the script runs.

The scripts provide timing signoff for TDM paths.

- If you see the bit file is not written out, check that the TDM fast clock is not failing.
- If you see the following critical warning, check that there are no missing constraints between the DUT and TDM.

SLL Opt may have caused timing violation in the DUT clocks. Bit file will be generated. However, the design might only be functional at runtime when clock frequencies are adjusted.

Working with Mixed-System Designs

To maximize hardware resources, you can set up the design on mixed systems. The following guidelines are for working with mixed designs that use a combination of HAPS-80 and HAPS-100 systems.

1. Follow these guidelines to launch and run the software:
 - Make sure to run the HAPS-100 version of the tool: protocompiler100.
 - Set the technology (-technology) to HAPS-100.
 - Use the UC flow for the design, because HAPS-100 designs require the unified compiler.
 - Do not use a mixed-language design. This is not currently supported for mixed hardware systems. UC supports mixed HDL with Verilog top. This is independent of HAPS-100/HAPS-80 systems.
2. Specify the hardware setup.
 - Set up the hardware and describe the connections in the TSS file.
 - Set up the clocks and resets.
3. Run the design through the normal flow and export the bit files.

- If you are using `haps_DTD_builtin` buffer type in HAPS-80 system, specify non-locked bins in the IDC file so that the DTD reset is replicated on all non-locked FPGAs of HAPS-80. For example:

```
-iice IICE_1 -dtd_reset_bins {FB2.uA}
```

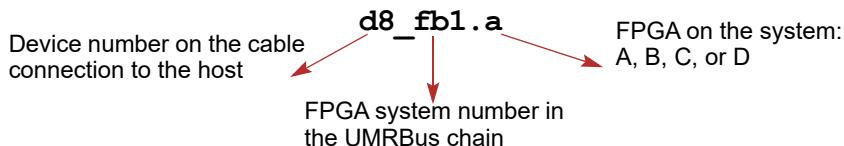
If you specify the locked bins you might get place-and-route error messages about black boxes or unsupported RTL constructs in the netlist.

- When partitioning the design which has CAPIMs instantiated, assign UMRBus 3.0 MCAPIM to HAPS-100 and UMRBus 2.0 to HAPS-80. See [Using UMRBus 3.0 with HAPS-100 Modules, on page 320](#) for details.
- For HSTDMD training, use the `proto_rt` command with the `-train` argument. See [Using proto_rt for HSTDMD Training, on page 307](#) for details.

4. Configure the hardware using generated targetsystem.tsd file using the cfg_project_configure command. This file can be found under exported runtime directory. See [Configuring Bit Files, on page 616](#) for details.
 - If you are using client-server communication, see [Debugger Configuration, on page 109](#) in Debugger User Guide for details.
5. Open ProtoCompiler Runtime after configuring the hardware. and debug the design as regular.
6. If you are using GSV, note these points:
 - To use GSV with HAPS-100, the verdi_mode and prepare_readback options must both be set to 1. You must use the HAPS-100 GSV flow for GSV designs; do not use the old flow.
 - There is a difference between HAPS-80 and HAPS-100 names in the ll files used at runtime. HAPS-80 names must be the physical FPGA names, while HAPS-100 names are the logical FPGA names from TSS. In the following example, the first two lines refer to HAPS-100, and the last two to HAPS-80:

```
/FB1.uA.ll -fpga FB1.uA  
/FB1.uB.ll -fpga FB1.uB  
/FB2.uA.ll -fpga d8_fb1.a  
/FB2.uB.ll -fpga d8_fb1.b
```

The physical FPGA name syntax for HAPS-80 is explained here:



Running in Batch Mode

Batch mode is a command-line mode in which you run scripts from the command line. You might want to set up multiple synthesis runs with a batch script. Batch scripts are in Tcl format. For more information about Tcl syntax and commands, see [Working with Tcl Scripts and Commands, on page 609](#).

This section describes the following operations:

- [Running Batch Mode with a Tcl Script, on page 603](#)
- [Queuing Licenses, on page 604](#)
- [Releasing the Tool License During Place and Route, on page 607](#)

Running Batch Mode with a Tcl Script

The following procedure shows you how to create a Tcl batch script for running compile and map operations. .

1. Create a Tcl batch script. See [Creating a Tcl Script, on page 611](#) for details.
2. Save the file with a tcl extension to the directory that contains your source files.
3. From a command prompt, go to the directory with the files and type one of the following as appropriate:

```
protocompiler -batch Tcl_script.tcl
```

The tool executes the script, running in batch mode. The results and errors are written to the log file. The tool also reports success and failure return codes.

4. Check for errors.
 - For source file or Tcl script errors, check the standard output for messages. On Linux systems, this is generally the monitor in addition to the stdout.log file; on Windows systems, it is the stdout.log/protocompiler.log file in the directory where it was run.

- For run errors, check the log file in the working directory. The software returns the following error codes if there are errors after a batch run.
 - 0 - OK
 - 2 - Logical error
 - 3 - Startup failure
 - 4 - Licensing failure. See [Queuing ProtoCompiler Licenses, on page 605](#) for more information.
 - 5 - Batch mode not available (Not applicable)
 - 6 - Duplicate-user error (server setups that do not permit multiple runs)
 - 7 - Project-load error (Not applicable)
 - 8 - Command-line error
 - 9 - Tcl script error
 - 20 - Graphic resource error
 - 21 - Tcl initialization error
 - 22 - Job configuration error
 - 23 - Parts error (Not applicable)
 - 24 - Product configuration error
 - 25 - Multiple top levels

Queuing Licenses

A common problem when running in batch mode is that the run fails because all of the available licenses are in use. License queuing allows a batch run to wait for the next available license when a license is on the server but not immediately available. You can specify either blocking or non-blocking queuing.

| Queuing Style | Tool Behavior |
|----------------------------|--|
| Blocking-style queuing | Waits until a license becomes available |
| Non-blocking-style queuing | Waits the specified length of time for a license to become available |

You can also queue DesignWare IP licenses, so that they can be used as they become available.

For details, see the following:

- [Queuing ProtoCompiler Licenses, on page 605](#)

- [Queuing Synopsys DesignWare IP Licenses](#), on page 606

Queuing ProtoCompiler Licenses

The following procedure describes how to specify blocking-style or non-blocking style queuing for synthesis licenses. You can specify the licensed features for queuing in an environment variable or directly in batch mode.

1. Consider these points when using queuing:
 - If blocking-style queuing is used, license checkout does not exit until a license becomes available.
 - There is no maximum wait time. Once initiated, the tool can wait indefinitely for a license.
 - If the server shuts down while the tool is waiting, you get a checkout failure.
 - When two licenses are required, queuing waits only until the first license becomes available (and not the second) to avoid holding a license unnecessarily.
2. Specify the list of licensed features you want to queue:
 - Specify a list of features to wait for using the `-batch`, `-licensetype` and `-license_wait` options. For example:

```
protocompiler -batch -license_wait 60  TclScript.tcl
```

See [ProtoCompiler Startup Commands](#), on page 12 in the *Command Reference* for details about the syntax.

3. To enable blocking-style queuing, do one of the following:

- Set environment variable `toolName_LICENSE_WAIT=1`.

`PROTOCOLRDX_LICENSE_WAIT=1`

```
PROTOCOLR_COMPILER_LICENSE_WAIT=1  
PROTOCOLR_RUNTIME_LICENSE_WAIT=1
```

- In batch mode, include a `-license_wait` command-line argument, as shown in the following examples:

```
protocompiler -batch -license_wait Tcl_script.tcl  
protocompiler_runtime debug -shell -license_wait  
Tcl_script.tcl
```

With blocking-style queuing enabled, the tool waits until the requested license becomes available. It generates the following message in the stdout.log or the Tcl window:

Waiting for license: *toolName*

4. To enable non-blocking-style queuing, do either of the following:

- Set environment variable *toolName_LICENSE_WAIT=waitTime* (*toolName* is the name of the FPGA synthesis tool and *waitTime* is the maximum wait time in seconds). For example:

```
PROTOCOL_COMPILER LICENSE_WAIT=180
PROTOCOL_COMPILER_RUNTIME LICENSE_WAIT=300
```

The *waitTime* value determines the maximum wait time, in seconds:

| WaitTime Value | Queuing Behavior |
|-----------------------|--|
| Undefined or 0 | Queuing off |
| 1 | Queuing on; wait indefinitely |
| >1 | Queuing on; wait up to the specified number of seconds |

- Include a *-license_wait waitTime* command-line argument when launching batch mode as shown in the following examples:

```
protocompiler -batch -license_wait waitTime Tcl_script.tcl
protocompiler_runtime debug -shell -license_wait waitTime
Tcl_script.tcl
```

When non-blocking-style queuing is enabled, the tool waits up to the maximum time limit specified for the license to become available. The tool generates the following message in stdout.log or the Tcl window:

Waiting up to n seconds for license: *toolName*

Queuing Synopsys DesignWare IP Licenses

In batch mode, the synthesis tool waits for an available IP license. When no license is available, the synthesis tool waits for either a specified time period or indefinitely. You can queue DesignWare IP licenses by adding a command line parameter to the tool invocation command.

The procedure to queue DesignWare licenses is described below; refer to [ProtoCompiler Startup Commands, on page 12](#) in the *Command Reference* for details about the syntax.

1. Take the following points into consideration:
 - The queuing mechanism does not take precedence over other Synopsys products such as the Design Compiler tool, and operates independently from their license queues.
 - Requested Synopsys DesignWare IP licenses are queued when the license is not immediately available.
2. To enable DesignWare IP license queuing in batch mode, use `-ip_license_wait` along with `-batch`. For example:

```
protocompiler -batch -ip_license_wait waitTime
```

waitTime is the number of seconds to wait for a license. If you specify 1, the synthesis tool waits indefinitely for the requested IP licenses.

The following examples illustrate command usage:

```
protocompiler -batch -ip_license_wait 60 myScript.tcl  
protocompiler_runtime debug -shell -ip_license_wait 60  
myScript.tcl
```

If an IP license is still not available when the wait period ends, the tool continues processing using any available license. An IP block without a requested license is processed as an error or a black box, if the option set `-dw_stop_on_nolic` command has a value of 0.

Releasing the Tool License During Place and Route

When invoking a third-party place-and-route tool from the Synopsys tool, you can let place and route continue to run even after exiting the tool, so that it does not consume a license. The tool lets you release the license and run place and route in batch mode.

1. To release the FPGA license, specify the following command:

```
toolName -batch -license_release
```

With this option specified, the tool checks in all prototyping tool licenses immediately after the place-and-route job is launched.

When licenses are released, you see the following message:

Exiting session due to -license_release option

Working with Tcl Scripts and Commands

The software uses extensions to the popular Tcl (Tool Command Language) scripting language to control synthesis and for constraint files. See the following for more information:

- [Using Tcl Commands and Scripts](#), on page 609
- [Generating a Tcl Script from the GUI](#), on page 609
- [Creating a Tcl Script](#), on page 611
- [Setting Number of Parallel Jobs](#), on page 612

Using Tcl Commands and Scripts

1. To get help on Tcl syntax, do any of the following:
 - Refer to the online help (Help->Tcl Help) for general information about Tcl syntax.
 - Refer to the *Command Reference Manual* for information about the run commands.
 - Enter `help *` in the tool window for a list of all the tool Tcl commands.
 - Enter `help commandName` in the Tcl window to see the syntax for an individual command.
2. To run a Tcl script, do the following:
 - Create a Tcl script. Refer to [Creating a Tcl Script](#), on page 611.
 - Run the Tcl script by entering `source Tcl_scriptfile` in the tool window.

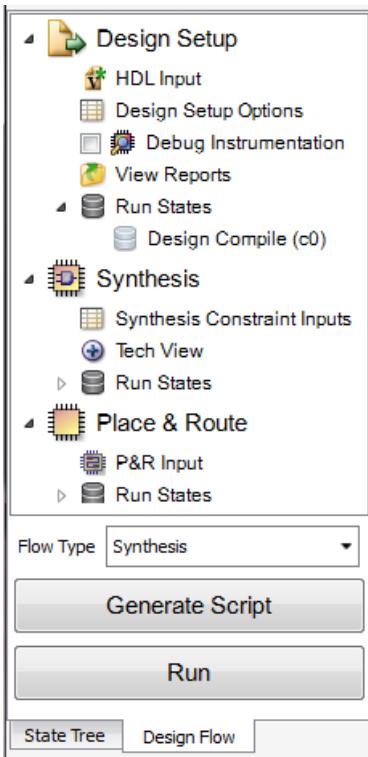
The software runs the selected script by executing each command in sequence. For more information about Tcl scripts, refer to the following sections.

Generating a Tcl Script from the GUI

The Design Flow tab in the GUI provides a convenient interface where you can generate a Tcl script based on the parameters you provide, instead of manually creating a script.

1. Create a root database.

2. Select the Design Flow tab (lower right of the tool window) and do the following:
 - For single-FPGA designs, set Flow Type to Synthesis. For multi-FPGA designs that must be partitioned, set Flow Type to Partition. The choices are reflected in the panel above the tab, which displays various phases and walks you through the flow steps needed to generate the script.
 - Click the first task (HDL Input), and add the source files in the tab that opens on the right side of the window. It is the standard GUI interface for entering source files.



- Similarly, walk through each of the steps in the order shown in the panel, adding options and constraints, and specifying run commands (Run States) as needed.
3. To generate a script, click Generate Script at the bottom of the panel.

The tool generates a Tcl script in the working directory, based on the options you set. It also displays the script in a tab on the right side of the tool window. Review the script.

4. Click Run to run the script.

Creating a Tcl Script

Tcl scripts are text files with a `tcl` extension. You can use the graphic user interface to help you create a Tcl script. Interactive commands that you use actually execute Tcl commands, which are displayed in the Tcl window as they are run. You can copy the command text and paste it into a text file that you build to run as a Tcl script. The following procedure covers general guidelines for creating a synthesis script.

1. Use a text file editor or select File->New, click the Tcl Script option, and type a name for your Tcl script.
2. Start the script by specifying the database command.
3. Add files using the `-src` or `-srcfilelist` argument to the `run compile` or `run pre_instrument` commands.

The files are added to their appropriate directories based on their file name extensions. Make sure the top-level file is last in the file list.

4. Add attributes and constraints as needed in `cdc` and `fdc` files.
5. Set other design synthesis controls with the `option set` command.
6. Include `run` commands to run through the various design phases.

For example: `run compile`, `run pre_map`, `run map`.

7. Check the script syntax.
 - Check case (Tcl commands are case-sensitive).
 - Start all comments with a hash mark (#).
 - Always use a forward slash (/) in directory and pathnames, even on the Windows platform.

Setting Number of Parallel Jobs

Some operations, like distributed processing, run jobs in parallel if there are licenses available. You can set the maximum number of parallel jobs by specifying the `max_parallel_jobs` variable as a command argument or by setting an option either directly from the Tcl window or in a Tcl file.

With multiple definitions, the option setting takes precedence over the command argument. A setting defined later in the flow takes precedence over one that was defined earlier. Details of the different methods to define the number of parallel jobs are described below:

1. To specify the maximum number of parallel jobs using the option command, do the following:

- Use this syntax to specify `max_parallel_jobs` with the option Tcl command:

```
option set -max_parallel_jobs value
```

Set the value based on the number of licenses you have available. With each license, you can run four jobs in parallel.

- Determine how you are going to issue the command. You can type it directly in the Tcl window or include it in a Tcl script or a Tcl file, like an options or setup file. If you specify it in a Tcl file, you must source the file. If you specify it in the Tcl window, the tool does not save the value, and it will be lost when you end the current session.

The tool applies the value globally to the design at the system level and does not apply to the individual FPGA runs. The maximum number of parallel jobs remains in effect until you specify a new value. When you set a new value, it takes effect immediately, going forward. For example, if you defined a value in a Tcl options file when you first compiled, but then typed in a value in the Tcl window at the map stage, the tool uses the value typed in the Tcl window going forward.

2. To set the number of parallel jobs as a command argument when you launch the tool, do the following:

- Specify it as an argument when you first start the tool from the command line:

```
protocompiler -max_parallel_jobs 8
```

If you do not specify this option, the default is 4.

- From the Tcl window, specify it as part of the launch protocompiler command when you implement partitioned FPGAs:

```
launch protocompiler -max_parallel_jobs 8 -script script1.tcl -script  
script2.tcl
```

This command is used to launch single_FPGA synthesis jobs in a multi-FPGA design.

- To allocate a different number of parallel jobs for different FPGAs, specify -max_parallel_jobs values before each individual -script options.

```
launch protocompiler -max_parallel_jobs 8 -script script1.tcl  
-max_parallel_jobs 10 -script script2.tcl
```

This command is used to launch a synthesis script with 8 jobs and a separate synthesis script with 10 jobs.

The number of jobs specified in a command argument can be overridden at any time by using the option set command to specify a new value.

You must specify the number of jobs with the launch command if you want to run the FPGA implementations in parallel, even if you already set the number of parallel jobs at a previous stage of the design. If you do not specify a value, the tool runs the FPGA implementations serially, instead of parallel processing them. The default is 1, which equates to four jobs in parallel, or eight jobs in parallel for area estimation. You can also override the setting specified in the launch command by setting an option later. See [Implementing Individual FPGAs, on page 426](#) for more information about implementing FPGAs.

Preparing to Run on the Hardware

Bit files are generated for the FPGAs after place and route completes. Once this is done, the bit files must be packaged with other necessary files into a configuration project is used to configure the HAPS system. The Confpro tool is used to create this project and then configure the HAPS system.

The following procedures briefly describe how to export the bit file and generate the configuration project. For details about configuring the system, refer to the *System Configuration Software Handbook* and the Confpro documentation.

- [Exporting Files for Runtime](#), on page 614
- [Configuring Bit Files](#), on page 616
- [Generating a Confpro Project](#), on page 617
- [Ensuring Hardware Bring-up](#), on page 621
- [Configuring HAPS Clocks](#), on page 624
- [Running HAPS Hardware Diagnostics](#), on page 626
- [Running Data Expansion in a Multi-User Setup](#), on page 627

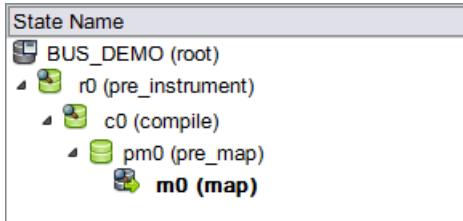
Exporting Files for Runtime

Once place and route is complete, export the files so that they can be used by the ProtoCompiler runtime executable, which is used to map the logic to the hardware setup and for debug. The following procedure describes how to set up the design for use with the hardware:

1. Start from the mapped database state of an FPGA after it has been placed and routed.

The design must be properly mapped to the HAPS hardware. See [Mapping to Hardware](#), on page 481 for more information, and consult the relevant HAPS system documentation.

If needed, use the appropriate database commands to go to the mapped database state. See [Loading and Navigating Database States](#), on page 62 for more information.



2. Ensure that all associated files, like IP files, are part of the design to be exported.
 - Run the import vivado *vivadoDir* command and specify the name of the Vivado directory where place and route was run.
 - If you skip this step, you can get an error message about missing files, if all the associated files are not imported back into the mapped state.
3. Export the files for ProtoCompiler Runtime.
 - To avoid errors, check that the following are in the database before exporting:

| Required File | Details |
|------------------------------------|---|
| Bit file for single-FPGA synthesis | The bit file might not be included in some scenarios; for example when you import a post-place DCP from a design that failed to route in order to try resynthesis after place and route. To make sure the bit file is in the database, run the import vivado command, which imports the bit file into the database. The bit file might not be required in the multi-FPGA flow, if all the FPGAs are not used. |
| instr.db file | Debug point database from debug_eco, which is required for modifying the debug points. |

If the database does not include these files, you can get errors when you run `export runtime`.

- Specify this command to export the files:

```
export runtime -path outputDirectory
```

This command creates a directory with the files needed, including the bit file generated after the place and route. For more about working with this file, refer to [Generating a Confpro Project, on page 617](#).

- The prototyping tool automatically exports all necessary files, including the bit files, to the specified directory. It prints messages about the files that are exported in the console.

If you manually copy exported files to another location, you must copy all exported files. This includes encrypted files, which might be required for successful bring-up.

- You can now use the exported files to debug the product with `protocompiler_runtime`. See [Running the RTL Debugger, on page 681](#) for details.

4. Set up the required tools and hardware:

- Make sure you have the Confpro utility installed. You can download it from <https://solvnet.synopsys.com>. The access details are described in [Ensuring Hardware Bring-up, on page 621](#).
- Make sure you have the ProtoCompiler runtime software, which is a separate executable in the ProtoCompiler installation.
- Make sure all cables and daughter boards are connected.

5. From the ProtoCompiler tool, start the runtime executable to bring up the hardware and run debug:

`launch protocompiler_runtime options`

Alternatively, run the `protocompiler_runtime` executable from a shell window.

Configuring Bit Files

You can speed up the time required to configure bit files by using multiple CPU cores to configure them in parallel. Use parallel configuration with HAPS-100 setups, HAPS-80 setups, and mixed HAPS-100/HAPS-80 setups. It cannot be used with HAPS-70 setups.

Note that the command described below only speeds up bit file configuration; it does not affect the time taken by other operations, such as HSTDm training.

1. To configure bitfiles in parallel, use the runtime command below, where *n* is the maximum number of jobs to be run in parallel:

```
cfg_project_configure cfg0 targetSystem.tsd -jobs n
```

The default is 1, and 0 means there is no limit on the number of jobs.

2. Define the number of jobs based on the systems used.
 - Each HAPS-100 system in the setup is connected directly to the host computer, and can be configured as an independent task to run in parallel.
 - For HAPS-80 systems, parallelization is governed by how the HAPS-80 system is connected to the host. Each HAPS-80 chain is considered as one task (there is no parallelization within a HAPS-80 chain), but multiple HAPS-80 super-chains can be configured in parallel.

Example: Configuring Bit Files in Parallel

In the following example, the maximum number of parallel tasks possible is 7: five for the five HAPS-100 modules and one for each HAPS-80 chain.

- 5 x HAPS-100
- HAPS-80 chain 1: 5 x HAPS-80
- HAPS-80 chain 2: 3 x HAPS-80

Configuration time will be dominated by the time required for HAPS-80 chain 1, which will always be treated as a single task regardless of the degree of parallelization.

Generating a Confpro Project

The Confpro project is required to configure the HAPS system. The project contains information about FPGA bit files, HAPS voltages, PLL frequencies and so on. The recommended method is to create the project automatically from the HAPS ProtoCompiler output, because the TSS file contains all the information about the HAPS system. The project can also be created manually by inspecting a live and configured HAPS system.

See the following procedures ([Creating a Configuration Project From TSS Definitions, on page 618](#) and [Creating a Configuration Project from a Live HAPS System, on page 620](#)) for more information about each of these approaches.

Creating a Configuration Project From TSS Definitions

This is the recommended way to generate a Confpro project file, automatically from the TSS. Use this method instead of the manual method described in [Creating a Configuration Project from a Live HAPS System, on page 620](#).

You can do this from the command line or the GUI, as described below. Refer to the Confpro documentation for details of the Confpro commands.

1. Make sure the information to be exported is complete.
 - Make sure the TSS is complete. It must define HT3 voltages and clock source settings and the correct CDE chain for chained systems. The TSS must be complete because the Confpro project is generated from this source.

Here are some examples of system configuration information defined in the TSS commands:

```
board_system_create -interconnect -manual CON_CABLE_100_HT3
    -name conn_1 -connector {mb1.A6 mb1.B6} -voltage 1.8
board_system_configure -connect {umrbus_if.CDE_OUT FB1.CDE_IN}
board_system_configure -connect {FB1.CDE_OUT      FB2.CDE_IN}
```

- Ensure that all associated files are in the database to be exported.

From the ProtoCompiler tool, run the import vivado *vivadoDir* command and specify the name of the Vivado directory where place and route was run. If you skip this step, you might get an error message about missing files, if all the associated files are not imported back into the mapped state.

2. From the ProtoCompiler tool, export the TSS hardware setup and other information after place and route is complete:

```
export runtime -path runtimeDir
```

You can now generate the configuration project, either from the GUI (step 3) or from the command line (step 4).

3. To use the GUI to create the project file, do the following:

- Open the Confpro GUI. For example: confpro_gui
- From the Confpro GUI, select File->New->Create From TSS File.
- Choose the system type and project name.
- Select the TSS file generated previously. For example:

runtimeDir/system/targetsystem.tsd

The specified TSS file has a tsd extension, because it is a tool-generated version of the TSS. It is recommended that you use this generated tsd file instead of the TSS file, because the tool automatically searches the tsd file and no runtime directory is needed. In addition, a tsd file is always elaborated, whereas a TSS file could be generic, with auto connections.

- If you did not specify the frequency in the TSS file, specify it now.
- Save the project file.

The command creates a new directory called *projectName/designs*, which includes the Confpro project file (*project.conf*) created from tsd information, as well as the configuration file (*.cfg*) and bit file from the *runtimeDir* directory.

4. To use shell commands to create the project file, follow these steps:
 - Start the Confpro shell with the *confprosh* command.
 - Use the Confpro *cfg_open* and *cfg_project_create_from_tss* commands shown in the example below to generate the file. The example assumes the runtime directory is *myRuntimeDir*, and specifies the creation of a new directory, *myProjectDir/designs*, which includes the Confpro project file (*project.conf*) created from tsd information, as well as the configuration file (*.cfg*) and the bit file from the *myRuntimeDir* directory.

```
set handle [cfg_open haps80 .]
cfg_project_create_from_tss
$handle ./myRuntimeDir/system/targetsystem.tsd ./myProjectDir
```

- If you did not specify the frequency in the TSS file, specify it now with the *cfg_clock_set_frequency* command.
5. Once all the parameters have been set, configure the project by clicking Configure System in the GUI.

Creating a Configuration Project from a Live HAPS System

If you want to generate a Confpro project from a live HAPS system, use the steps described below. In this method, you use Confpro to load the bit files to the individual FPGAs, and manually enter various Confpro commands to configure the system. The preferred method to generate a Confpro project is to use the TSS file ([Creating a Configuration Project From TSS Definitions, on page 618](#)).

You can use either the GUI or shell commands to generate a project from a live HAPS system. For details, refer to the Confpro documentation.

1. Set up the hardware system, and make sure all cables and daughter boards connected.
2. Start the Confpro utility.

To start the tool in standalone mode, do so from the host computer for the HAPS system. You can use either the `confpro_gui` (GUI mode) or `confprosh` commands (Tcl shell mode). You can also start Confpro from the prototyping tool with this command:

```
protocompiler_runtime board Bringup [-shell]
```

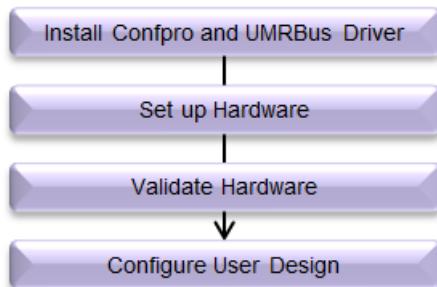
3. Set up the project based on the HAPS system and configure the bit files.
 - Either run the `cfg_project_create` shell command or select File->New->Scan platform to scan the active hardware and specify setup parameters in a template configuration file (`project.conf`). Alternatively, specify the parameters directly in the GUI to create the project file.
 - Manually specify the parameters needed for configuration using commands like `cfg_clock_set_enable`, `cfg_clock_set_frequency`, and `cfg_ht_set_vcc`. Make sure to specify a project name, a directory, the bit/bin files for each FPGA, clocks and resets, and voltages for the HapsTrak 3 connectors. The `cfg*` commands used to specify the setup parameters are described in the *System Configuration Software Handbook* and the Confpro documentation. See [Exporting Files for Runtime, on page 614](#) for information about configuring the bit file.
 - Load the bit files to the individual FPGAs.

Confpro automatically converts the bit files to bin files. It creates a `project.conf` file that includes the aliases of the bit/bin files, and it also copies these files and the `..cfg` files to the working project directory (designs).

4. Save the project.
5. Check that you are using the correct binaries for the FPGAs.
6. Once all the parameters have been set, configure the system by clicking Configure System in the GUI.

Ensuring Hardware Bring-up

The following figure summarizes what you need to do for hardware bring-up, and to set up the hardware to run debug. This is only intended to be an overview of board bring-up; for details about the steps refer to the appropriate documentation. The last two tasks listed in the flow can overlap, although they are separated here to first validate the hardware setup by itself before configuring the user design and validating it with the hardware.



1. Download the Confpro and UMRBus driver software from <https://solvnet.synopsys.com>.
 - Log in to the SolvNet® database, then click Documentation, and select HAPS from the alphabetized product list.
 - Go to Software->System Configuration Software & UMRBus Driver and download the platform-specific version of Confpro.
 - Download the platform-specific version of the UMRBus driver from the same location.
 - Install both tools, following the instructions in the accompanying README files.
2. Start the Confpro software from the host computer for the HAPS system, using one of these methods:

- In standalone mode, using the `confpro_gui` or `confprosh` commands to bring up the GUI or Tcl shell for the configuration commands, respectively.
- From the runtime executable of the prototyping tool:

```
protocompiler_runtime board_bringup
```

3. Set up the hardware.

- Exercise all ESD precautions when handling the hardware.
- Connect up the hardware so that the setup matches the hardware definition in the TSS file. Hardware setup includes the adapter board, daughter boards, cables, and UMRBus, in addition to the HAPS system itself.
- Check cabling between board systems and daughter boards to ensure that they are connected correctly and properly seated.
- Check the orientation of clock cables, and make sure that `clock_right` and `clock_left` are set up correctly for your design.

For details about setting up and connecting the hardware, refer to the documentation for the hardware you are using. The documentation is available on SolvNet, at the HAPS location mentioned in step 1.

4. Validate the hardware setup without the user design.

- Start the runtime executable:

```
protocompiler_runtime board_bringup [-shell]
```

You can use the shell or the graphical interface to enter the commands or the shell. For details about the bring-up utilities and the command syntax, see [Running the Board Bring-up Utility, on page 36](#) in the *Debugger User Guide* and [protocompiler_runtime Startup Commands, on page 15](#) in the *Command Reference Manual*, respectively.

- Select the UMRBus PCI/USB device.
- Specify the board ID and board type either in the GUI or with the `haps settings` command. For syntax details, see [haps, on page 52](#) in the *ProtoCompiler Debugging Environment Reference Manual*.
- Use `haps` commands to run tests to check the hardware setup.

| To ... | Use |
|---|--|
| Check board status and board type | <code>haps board</code> <code>haps boardtype</code> |
| Validate the hardware setup against the TSS | <code>haps tssgen</code> |
| Verify UMRBus access to the FPGA | <code>haps run umr_check</code> |

For details about using `haps tssgen`, see [*Validating the TSS Against the Hardware, on page 226*](#).

- Validate the HSTDm links.
5. Set up a Confpro project to configure the user design.
 - For overviews of the steps to follow, see [*Exporting Files for Runtime, on page 614*](#) and [*Generating a Confpro Project, on page 617*](#).
 - Check that you are using the correct binaries for the FPGAs. Confpro automatically converts the bit files to bin files. It creates a project.conf file that includes the aliases of the bit/bin files, and it also copies these files and the ..cfg files to the working project directory (designs).
 - Set configuration parameters. For details about configuring the system and the cfg commands for doing so, see the Confpro documentation.
 - In the GUI, click Configure System.
 6. Validate the user design with the hardware setup.

See [*Validating the User Design with the haps Command, on page 623*](#) for details.

Validating the User Design with the haps Command

Use the following procedure to check the HT3 cables and qualify them, and check that the I/O voltages, and clock settings are correct. Validating the user design ensures that it will run smoothly when you bring up the hardware.

The tests described here use `haps` commands. For more about this command and the tests run, see [*haps, on page 52*](#) in the *HAPS ProtoCompiler Debugging Environment Reference Manual*.

1. Run clock check with `haps run clock_check`.

This test configures the FPGAs using a test design and does not modify the user clock settings.

- Check that the clocks configured by the Confpro project are correct, and that the onboard FPGA clocks fall within the frequency range for the board system.
- For chained systems, check that there are no missing CDE cables between systems.
- Validate clock frequencies in the `hapstest.log` file. Check the report for out-of-range results.

2. Check the connectivity with `haps run con_check`.

You can run this test at any time. The user design is not removed from the hardware.

- First run the `export runtime` command in the prototyping tool, to generate a `connectivity.tcl` file.
- Edit the file so that FPGA references use the `fb*` syntax instead of `mb*`. Copy the file to your working directory.
- Return to Confpro, run `haps run con_check`, and check the I/O voltages against the input `connectivity.tcl` file.
- Check that the HT3 cables are present and correctly seated.

3. Test the HT3 connectors and cables with `haps con_speed` to validate the HT3 channels.

This check configures a test design onto the system. It uses LVDS data to transfer between FPGAs and check the electrical characteristics of the HT3 connections.

4. Check the HSTDm channels with `haps hstdm_report`.

You can run this test at any time. This test does not remove the user design from the hardware, nor does it change user settings.

Configuring HAPS Clocks

Use the following sequence of Confpro commands to configure HAPS clocks. There are some differences between HAPS-80 and HAPS-100, which are described in the subsequent table.

Except for enabling or disabling specific outputs, which can be done at any time, run the clock configuration commands before the FPGAs are configured.

1. Reset registers to the PLL-specific default values using one of the commands shown in the example:

```
cfg_clock_set_default cfg0 fb1 pll1  
cfg_project_clear
```

If you use the clear command, it automatically resets the defaults.

2. Calculate and write the PLL parameters for the desired frequency.

```
cfg_clock_set_frequency cfg0 fb1 pll1_1 1234  
cfg_clock_set_frequency cfg0 fb1 pll1_2 2345
```

3. Enable the output:

```
cfg_clock_set_enable cfg0 fb1 pll1_1 1  
cfg_clock_set_enable cfg0 fb1 pll1_2 1
```

This command is optional for HAPS-100 designs. See the table below.

4. Finalize the frequency plan and synchronize the outputs:

```
cfg_clock_calibrate cfg0 fb1 pll1
```

This command is optional for HAPS-100 designs. See the table below.

5. Disable an individual output to protect free-running design parts during FPGA configuration:

```
cfg_clock_set_enable cfg0 fb1 pll1_1 0
```

This command is optional for HAPS-100 designs. See the table below.

6. Enable or disable a specific output:

```
cfg_clock_set_enable cfg0 fb1 pll1_1 1 | 0
```

This command is optional for HAPS-100 designs. See the table below.

The following table lists some differences in the implementation of `cfg_clock*` commands, between the HAPS-80 and HAPS-100 systems.

| Command | HAPS-100 | HAPS-80 |
|-------------------------|--|---|
| cfg_clock_set_default | Resets the PLL default value. The default is also reset if you use the cfg_project_clear command. | |
| cfg_clock_set_frequency | Calculates the final frequency plan for all outputs and enables the output. Applies a soft reset for all outputs. | Calculates a preliminary frequency plan. The frequency could be wrong if the output is already enabled. |
| cfg_clock_set_enable | Optional because the outputs are already enabled with cfg_clock_set_frequency. When enabled (1), sets the output enable bit. When disabled (0), sets the output enable bit to 0. In both cases, it does not interfere with other outputs. | The frequency plan is not final and there might be a wrong frequency on the output. When enabled (1), sets the output enable bit. When disabled (0), sets the output enable bit to 0. In both cases, applies a soft reset for all outputs. |
| cfg_clock_calibrate | Optional. Applies a soft reset for all outputs. | Calculates the final frequency plan based on all enabled outputs. Applies a soft reset for all outputs. |

Running HAPS Hardware Diagnostics

You can run diagnostics on the HAPS hardware from the runtime environment.

1. To see information about each diagnostic test, specify this command in the runtime environment:

```
% haps help <name_of_the_test>
```

2. To run a diagnostic test, specify it at the console.

You can specify it from the command line or put the commands in a Tcl file. For HAPS-100, the following tests and diagnostic checks are available:

- `clock_check`
Reports clock frequency at the user FPGA after the hardware is configured with the TSS.
- `reset_check`
Checks that every user FPGA receives the reset signal.
- `con_electrical_check`
Sends a static pattern to check electrical connections for given cables defined in the TSS.
- `con_check`
Validates the VCCO enabled on the HT3 connector in the TSS, and checks that the cable position matches the TSS.
- `ht3_loopback_check`
Checks connectors for bad traces and bent pins; finds bad cables by connecting on loose end (stand alone or in the setup).
- `DDR4_selftest`
- `fixed_interconnect_check`
- `ht3_cable_reach_check`

Use this command to run diagnostics from a Tcl file:

```
protocompiler100_runtime board_bringup -batch -tcl file.tcl
```

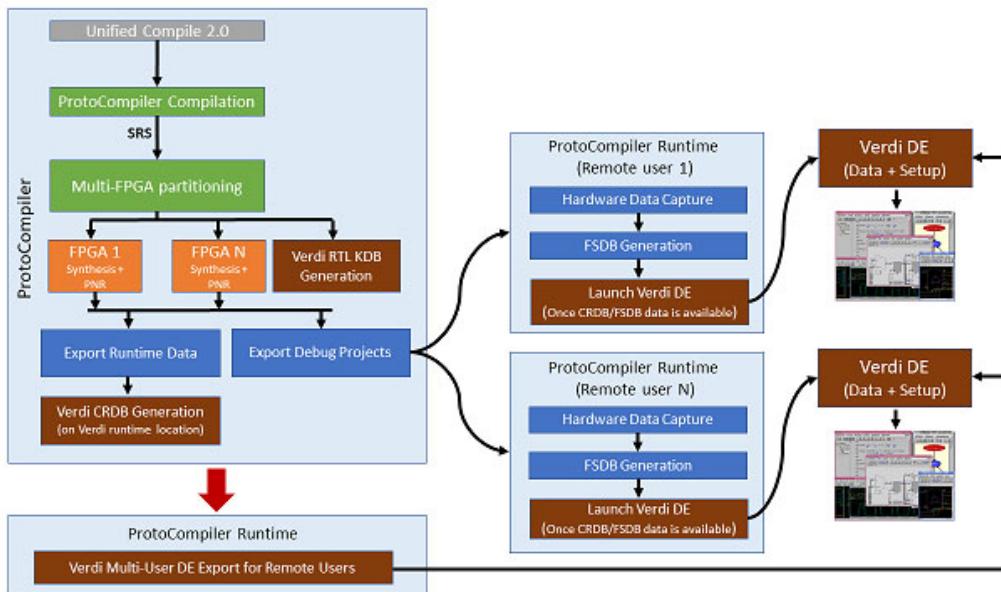
This is an example of the diagtest.tcl file for `ht3_loopback_check`:

```
haps settings { PORT_NAME /umr3usb1/mod-X007209 }
haps board FB1
haps boardtype HAPS-100
puts {##HAPS Diagnostics Test_BEGIN}
haps run ht3_loopback_check ./connector_list.txt
    /umr3usb1/mod-X007209 loopback_test.log
puts {##HAPS Diagnostics Test_END}
exit
```

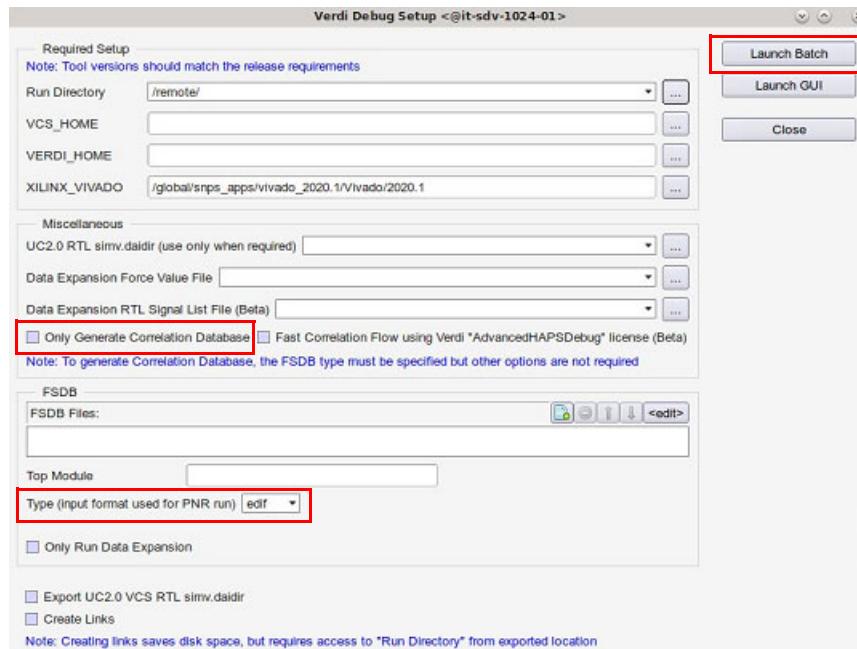
Running Data Expansion in a Multi-User Setup

You can run data expansion (DE) in large-scale multi-user setup, where one host machine is connected to multiple HAPS systems and multiple users access these systems from various locations. The correlation data base (CRDB) and the data expansion (DE) setup are exported to remote machines.

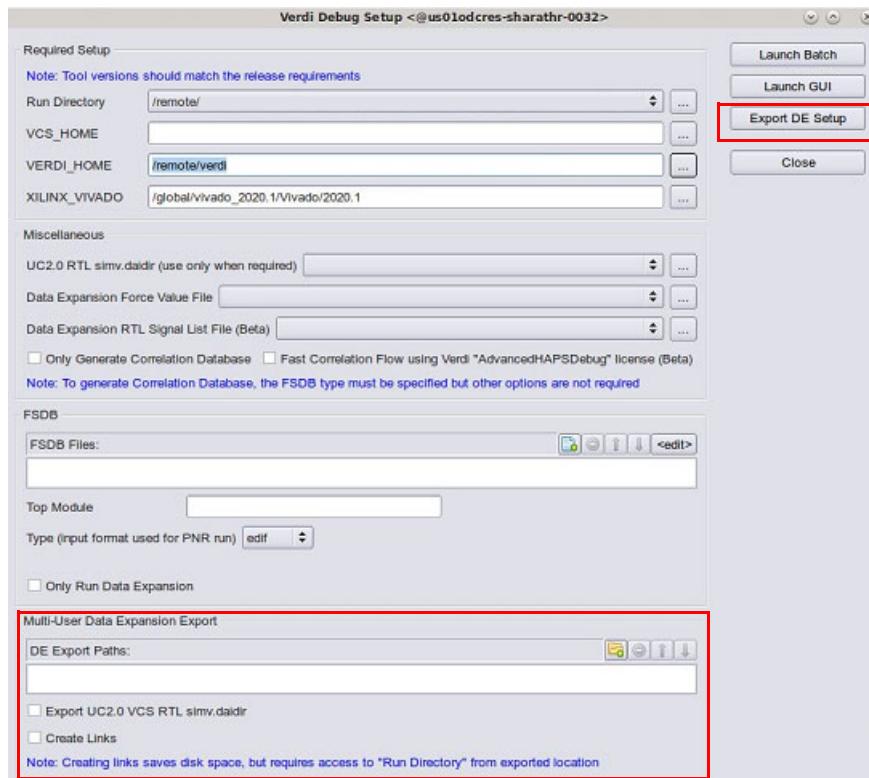
The following diagram shows the HAPS-Verdi Flow with Multi-User Verdi Data Expansion and Debug:



1. ProtoCompiler Primary user:
 - Generates CRDB using ProtoCompiler:
Run Launch Batch with Only Generate Correlation Database selected.



- Adds the directory paths to the Multi-User Data Expansion Export using the ProtoCompiler Runtime tool. The setup and data for multiple users are exported to this path.



This can also be done in ProtoCompiler Runtime batch mode using the following commands:

```
launch_verdi -run_dir <PrimaryUserVerdiDir> -crdbgen
export_verdi_de -from <PrimaryUserRunDirPath> -to
<RemoteUser1ExportPath> -to ...
```

- Exports the DE setup to the given paths using the Export DE Setup button.
2. Remote users then use the exported locations and launch ProtoCompiler Runtime > Verdi Debug:
 - Sets up run directory path to their exported location
 - Selects Only Run Data Expansion
 - Clicks Launch GUI to bring up Verdi debug environment

To ensure compatibility, use the same VCS_HOME that was used to generate the UC2 database and the same VERDI_HOME that was used to generate the CRDB. Change in versions can cause incorrect results during data expansion. All users must have access to RTL simv.daidir directory from their location.

See the following sections for more details:

- [Integrated Debugging with the Verdi Software](#)
- [export_verdi_de](#) in the FPGA Prototyping Command Reference Manual
- [launch verdi](#) in the FPGA Prototyping Command Reference Manual

CHAPTER 6

Instrumenting and Debugging Designs

The following topics provide details about the debug flow, which adds instrumentation and debugging steps to the basic implementation flow.

- [Evaluating Debug Methodology Choices](#), on page 634
- [The Instrumentor-Debugger Flow](#), on page 638
- [Instrumenting the Design for Debug](#), on page 641
- [Instrumenting with Unified Compile](#), on page 672
- [Verifying UC Designs](#), on page 677
- [Running the RTL Debugger](#), on page 681
- [Working with Deep Trace Debug](#), on page 695
- [Running Deep Trace Debug \(DTD\)](#), on page 702
- [Running Multi-FPGA Built-in DTD \(HAPS-80\)](#), on page 711
- [Using a Debug Hub with SATA \(DTD2\)](#), on page 716
- [Setting up Hardware for DTD2 with HAPS-70](#), on page 731
- [Using a Debug Hub with QSFP Connectors](#), on page 740
- [Using BRAM for Debugging](#), on page 743
- [Running Incremental Debug for ECOs](#), on page 748
- [Using Global State Visibility \(GSV\)](#), on page 753
- [Integrated Debugging with the Verdi Software](#), on page 792

Evaluating Debug Methodology Choices

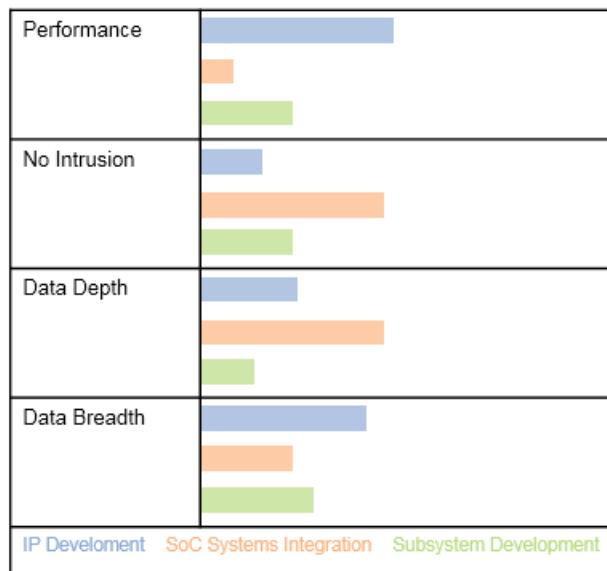
Debug is a complicated process because there is no one approach that fits every debug need. Designs are complex, there are many sources of debug information, and debug resources are scarce. The chosen debug approach is determined by the mix of performance requirements, debug visibility needs, and available hardware resources for that design. Different approaches are required to match different prototyping needs.

For a discussion of various aspects of the debug landscape, see these topics:

- [Debug Factors](#), on page 634
- [HAPS ProtoCompiler Debug Functionality](#), on page 635

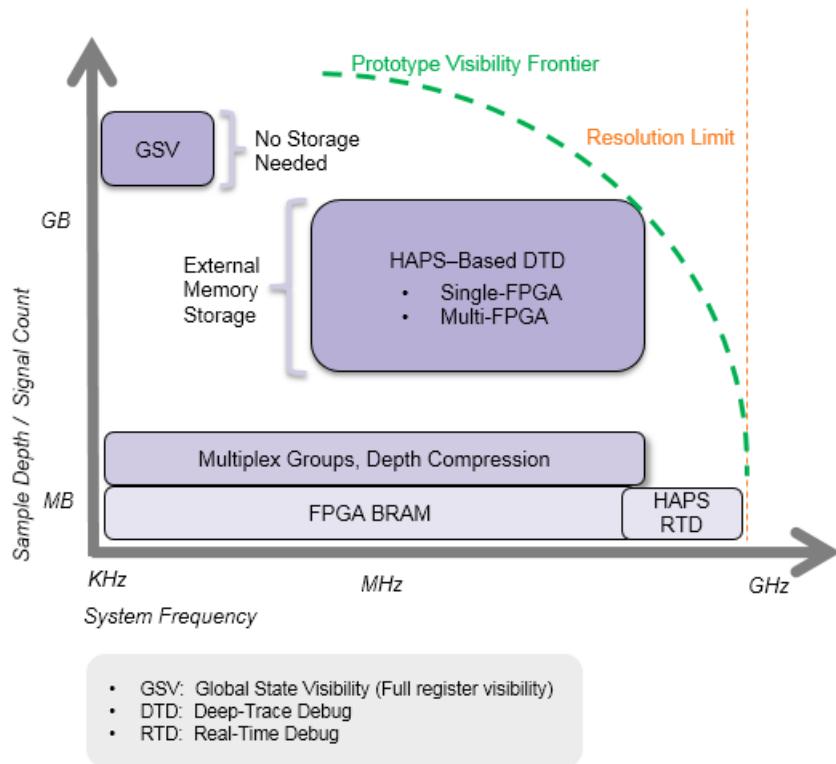
Debug Factors

There is a wide variance in the relative importance of debug factors, according to the type of prototype development. For example, IP subsystem development prioritizes data breadth (coverage) and performance over non-intrusiveness, whereas SoC system integration puts a premium on non-intrusiveness and depth of data coverage.



HAPS ProtoCompiler Debug Functionality

To address this array of debug priorities, the HAPS ProtoCompiler software provides multiple tools that address different parts of the debug spectrum. The following figure is a relative summary of the debug functionality, based on two of the criteria, performance and sample depth.



The following table lists other points of comparison for various debug methodologies and contains links to the information:

| Interactive | Not Interactive |
|-------------|---|
| Real-Time | RTD Using Real-time Debugging, on page 744 |

| Interactive | Not Interactive |
|---|---|
| Post-Processing | |
| GSV: <ul style="list-style-type: none"> • Single/Multi-FPGA • Wide, emulator-like visibility • No memory storage Using Global State Visibility (GSV), on page 753 | BRAM: <ul style="list-style-type: none"> • Single/Multi-FPGA • Broad coverage • Uses internal memory: BRAM resources on the FPGA • Fast Using BRAM for Debugging, on page 743 |
| | Built-in DTD (DTD4): <ul style="list-style-type: none"> • Minimally intrusive • Uses internal memory: built-in DDR3 (HAPS-80) • Large visibility window Running Multi-FPGA Built-in DTD (HAPS-80), on page 711 |
| | Hub-based Debug (DTD2 and DTD3): <ul style="list-style-type: none"> • Single/Multi-FPGA • Minimally intrusive • Requires additional HAPS-DX7 system for debug hub • Large visibility window Using a Debug Hub with SATA (DTD2), on page 716 Using a Debug Hub with QSFP Connectors, on page 740 |
| | External DDR3 or DDR4 daughter board (DTD): <ul style="list-style-type: none"> • Single FPGA • Minimally intrusive • Requires additional daughter board resources • Large visibility window Running Deep Trace Debug (DTD), on page 702 |

As the table shows, memory is another key factor that affects the choice of debug methodology, especially for scenarios where depth of coverage is important. Memory requirements vary widely for different debug schemes;

global state visibility (GSV) requires no memory storage, while deep-trace debug (DTD) schemes are memory-intensive, requiring either built-in system hardware resources or additional external memory.

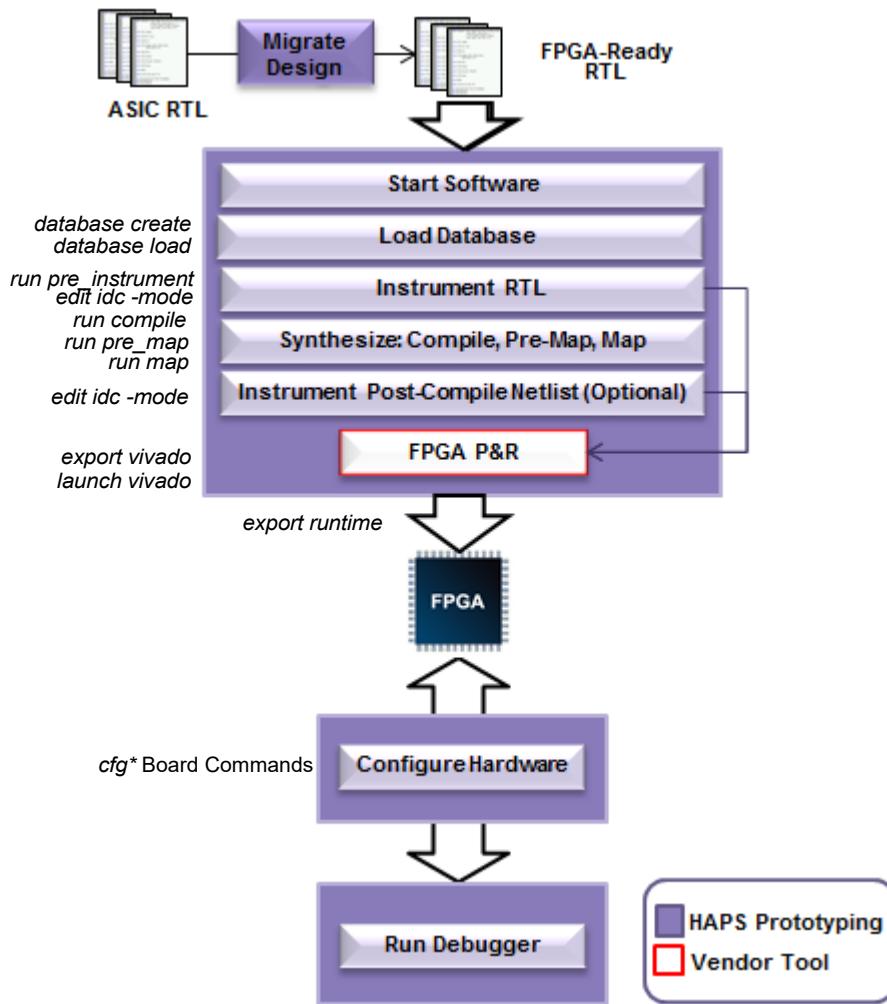
Debug memory can be categorized as internal or external. Internal memory refers to BRAM memory that is part of the resources available in the FPGA modules. External memory refers to an additional memory module that is added to an FPGA for memory expansion, like daughter cards or a debug hub. HAPS-80 systems include an external memory module, to support built-in deep trace debug.

The Instrumentor-Debugger Flow

The prototyping tool includes functionality for instrumenting the design and then debugging the RTL. RTL debugging ensures functional correctness that a post-synthesis debugger cannot guarantee. This is because synthesis optimizations cause gate-level netlists to significantly differ from the functional RTL description, making it hard to trace back bugs to the RTL.

The Instrumentor-Debugger flow is implemented in two stages. You select the signals you want to observe at the HDL level, by adding signal probes and triggers directly into the HDL source files. Later in the design cycle when you debug the design, you check the instrumentation and functionally verify the RTL code. As with simulation, the control of hardware triggers and annotation of captured data offer a comprehensive design view and easy probing. To that, the built-in functionality adds speed and accuracy of results, which are areas where simulation is limited.

The following figure shows the various phases of this debug flow and the main commands associated with each stage.



For details about the steps shown, follow these links:

- [Instrumenting and Debugging Designs](#), on page 633
- [Converting ASIC Designs to FPGA](#), on page 33
- [Working with a Design Database](#), on page 58
- [Compiling the Design](#), on page 77
- [Instrumenting the Design for Debug](#), on page 641

- [Running Pre-Map](#), on page 474
- [Mapping the Design](#), on page 490
- [Analyzing Results](#), on page 512
- [Running Place and Route](#), on page 576
- [Exporting Files for Runtime](#), on page 614
- [Running the RTL Debugger](#), on page 681
- [Preparing to Run on the Hardware](#), on page 614

Overview of the Instrumentor-Debugger Flow

The following steps provide an overview of the flow shown in the previous diagram. For syntax details for the commands mentioned, refer to [Shell Command Reference](#), on page 17 in the *Command Reference* manual.

1. Start the tool using the protocompiler command, and load the database.
2. Launch the instrumentor and instrument the design, as described in [Instrumenting the Design for Debug](#), on page 641.

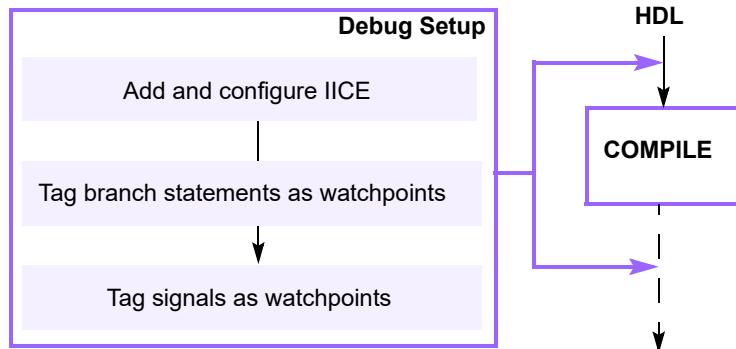
For designs that have already been instrumented, open the idc file.
3. Compile the instrumented design.
4. Implement the FPGA by synthesizing, placing, and routing it (run pre_map, run map, and run par_explorer/run par commands).

Optionally, you can instrument the post-compile netlist.
5. Export the bit file with the export runtime command.
6. Debug the design as described in [Running the RTL Debugger](#), on page 681.
 - Launch the runtime executable.
 - Start the debugger with the protocompiler_runtime debug command.
 - Debug the design.
7. To verify the design using the hardware, follow the steps described in [Ensuring Hardware Bring-up](#), on page 621.

Instrumenting the Design for Debug

The first step to prototyping with debug is to set up the design and mark signals for debugging. Do this by adding debug point logic, which is used to run on-chip debugging later, after programming the FPGA. It is a good practice to add debug point logic early in the cycle and reserve the resources needed, instead of doing it after the design is in place. Incorporating debug points as part of the initial phase reduces turnaround time. This step allows you to pipe clean the flow and identify logic, memory, timing and other limitations early in the cycle.

You can add instrumentation to a pre-compiled netlist or to one that has already been compiled. This is an overview of the details of flow:



For step-by-step procedures, see the following topics:

- [Instrumenting the Design Before Compiling](#), on page 642
- [Instrumenting the Design After Compiling](#), on page 648
- [Defining IICE Parameters](#), on page 651
- [Selecting Buffer Type](#), on page 657
- [Specifying IICE Clocks](#), on page 659
- [Making Incremental Instrumentation Changes](#), on page 663
- [Making Incremental Changes in UC Instrumentation](#), on page 665
- [Instrumenting Multiple Clocks in a Single IICE](#), on page 668
- [Capturing Startup Errors with Pre-Armed Triggers](#), on page 670

Instrumenting the Design Before Compiling

There are pros and cons to adding debug points to the design before compiling, as opposed to after compiling. Marking signals before compiling offers more visibility into the design and lets you implement complex trigger and muxing options. However, some logic, like generate statements, might not be visible, and the design and available resources are probably not stable. Further, adding debug points at this stage modifies the HDL, because it requires that extra logic be created for debug.

The following procedure is an overview of how to add debug points to a design before it is compiled. For a complete description of the functionality and commands, refer to the online help built into the GUI.

1. Start with a root database, and create a database state for instrumentation by using this command:

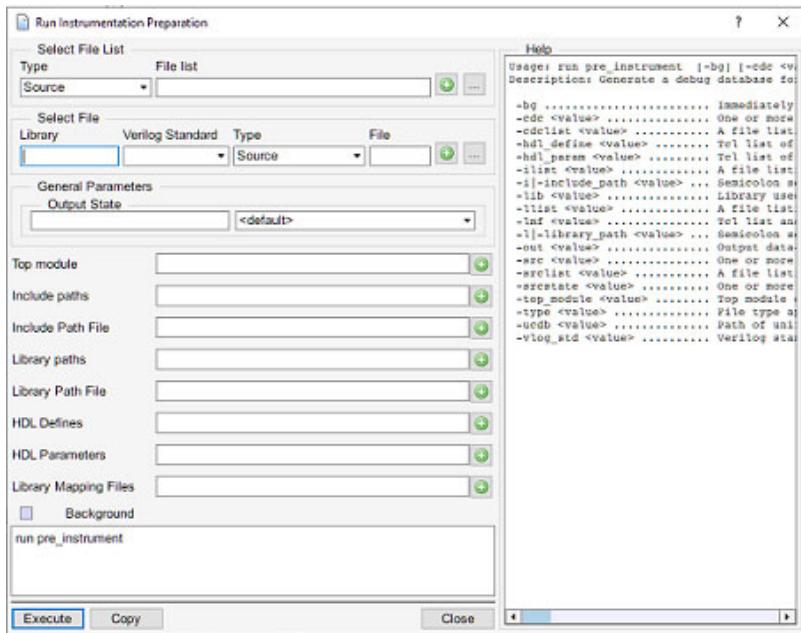
```
run pre_instrument -srclist sourceFileList
```

For a multi-FPGA design you instrument the design at the top level, before partitioning it.

You must specify the source files for the design at this point because you have not yet compiled the design. Optionally, you can also specify constraints and options files. See [Specifying Source Files \(Standard Compiler\), on page 97](#) for information about adding design files.

The run pre-instrument command creates the database and prepares the design for all the signals that can be instrumented. For the syntax, refer to [run pre_instrument, on page 149](#) in the *Command Reference Manual*.

You can also run this command from the GUI. You must specify the source files in the dialog box that opens.



The tool compiles the design and creates an RTL debug state that is ready to be instrumented, with the default name r0.

root → *run pre_instrument* → r0

2. Open the instrumentor GUI by typing the following command:

`edit idc -mode gui filename`

You must specify a file name to either create a new one, or to open an existing instrumentation (idc) file. Use the -mode gui option to open the GUI interface.

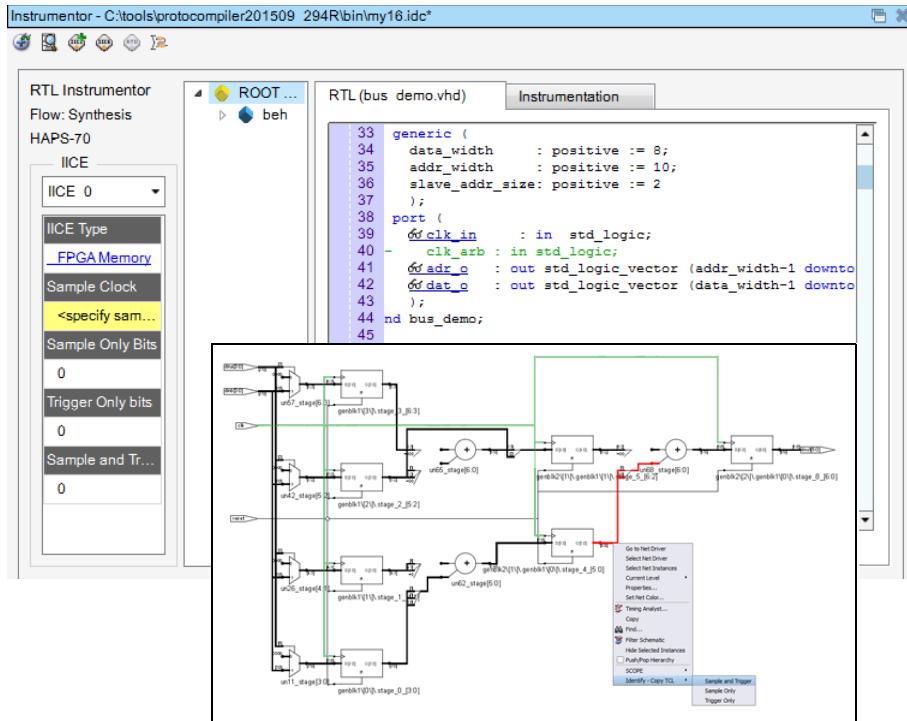
3. Instrument the RTL design by setting watchpoints and breakpoints that you want to trigger and sample when you debug the design.

The following instructions provide a high-level overview. For more comprehensive information, refer to the help built into the instrumentor.

- Set watchpoints. A watchpoint is a signal or node that is connected to the instrumentation logic that can be used for sampling or as a trigger point. Navigate the RTL hierarchy and tag nodes as

watchpoints by clicking on them. You can also open the schematic and tag watchpoints by right-clicking the logic you want to tag.

The addition of probes does not affect timing performance. The increase in design size is minimal.



- Tag RTL branch statements as breakpoints. A breakpoint is an RTL control flow statement (IF, THEN, CASE) that triggers sampling.

For some guidelines on instrumenting the design and debug planning, see [Recommendations for Debug Planning](#), on page 647.

```

42
43 process ( curr_state, req1, req2 )
44 begin
45   grant1 <= '0';
46   grant2 <= '0';
47
48   case (curr_state) is
49     when st_idle1 =>
50       if ( req1 = '1' ) and ( req2 = '1' ) then
51         next_state <= st_grant2;
52       elsif ( req2 ) then
53         next_st: <= Sample and trigger
54       elsif ( req1 ) then
55         next_state <= st_grant1;
56       else
57         next_state <= st_idle1;
58       end if;
59     when st_idle2 =>
60       if ( req1 = '1' ) and ( req2 = '1' ) then
61         next_state <= st_grant1;

```

- Add IICE™ (Intelligent In-Circuit Emulator) and communication blocks for probe and communication logic to trigger and sample the design. You can add multiple IICES to handle multiple clock domains.
- Save the design. The tool writes out an idc file with information about the instrumented design. You can open this file on subsequent runs with the edit idc command.

See the scripts provided with the tool for examples of simple scripts you can use to automate instrumentation tasks. Also refer to the example in *Instrumentation Script Example*, on page 647.

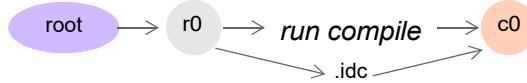
4. To pre-configure and pre-arm triggers, follow the steps described in *Capturing Startup Errors with Pre-Armed Triggers*, on page 670.
5. Specify the memory to be used as the sample buffer for debugging.

You can set the buffer type from the IICE Sampler tab in the instrumentor GUI or by typing the iice sampler command. See *Selecting Buffer Type*, on page 657 for more information. You can use either onboard memory or external DDR3 memory, according to your hardware setup and the debug mode.

6. Compile the instrumented design: run compile

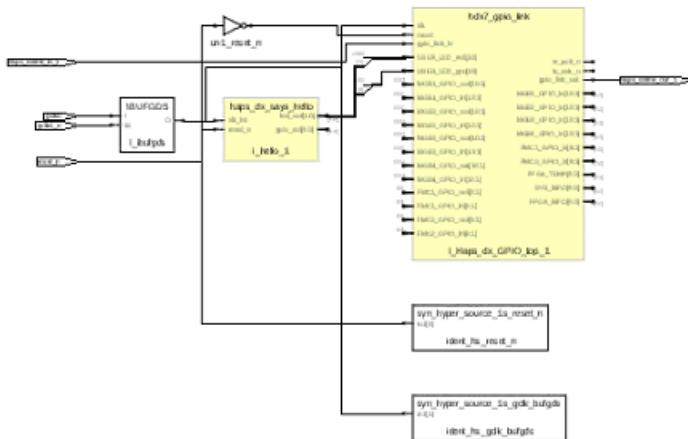
The run compile command automatically includes information from the idc file when it compiles the design. See *run compile*, on page 132 in the *Command Reference* for details about the command syntax.

Optionally, enable the compiler to run incrementally by setting the option incremental_debug to 1 and entering database apply_state -idc in the command line. For more information, see [incremental_debug under List of Options, on page 82](#) in the Command Reference and [Incremental Compile, on page 672](#).



7. Analyze the instrumented design.

You can view the data in the schematic with the view schematic command, which lets you trace nodes back to the original RTL. You can also open the schematic by clicking the plus icon (⊕) in the GUI.



You can also instrument signals in the schematic. The schematic instrumentation can be in addition to the RTL-based instrumentation that was done earlier, or as an alternative to it.

See [Instrumenting the Design After Compiling, on page 648](#) for information on editing and adding instrumentation to a compiled netlist.

8. Continue with the rest of the flow as usual.

For single-FPGA debug, continue with the synthesis flow (pre-map and map). For multi-FPGA debug, continue with the partitioning flow.

Recommendations for Debug Planning

The following tips provide recommendations on instrumenting your design:

- Proactively plan for debug visibility
 - Plan for debug when you plan the clocks, resets, and I/Os. In particular, consider the clock structure and how it maps to HAPS global clocks, MMCMs and PLLs for clock control and synchronization, and reset synchronization and control.
 - Plan for debug with incremental logic changes to an existing design, as when new IP is added.
- Mark signals for debug
 - Create separate .idc files for each block
 - Use multiple IICEs to debug different clock domains
 - Group signals in each IICE using mux groups, and selectively debug
- Consider memory depth
 - For signals that require a large sample window, use external memory, with a Deep Trace Debug buffer. With this methodology, there is a trade-off for the sample frequencies that can be supported. For further details, see the sections that describe deep trace debugging.
 - For designs that require a high sample clock (e.g. interface IP) consider using BRAM or real time debug (RTD). BRAM uses block RAM in the FPGA, so is best suited for shallow sample windows. Use RTD with an external logic analyzer if you need a much larger sample window.

Instrumentation Script Example

You can set up a script to instrument your design. This is one example of a script that instruments the design without starting the GUI. For others, refer to the debug examples included in the tool installation.

```
#Open the debugger project
set prj_path myPath/debug/debug.prj
project open $prj_path

#Set the focus to i250 debugger logic
iice current iice_dtd
```

```
#Set the sample depth, 20 signals instrumented
iice sampler -iice iice_dtd -sampledepth 500000

set keystroke Y
set i 0

while { $keystroke eq Y } {
    #Time before running the Run command
    set sys_time_start [clock seconds]
    puts "Time before RUN command: [clock format $sys_time_start
        -format %H:%M:%S]"

    #Run the debugger
    run -iice iice_dtd -wait

    set sys_time_end [clock seconds]
    set sys_time_difference [expr $sys_time_end - $sys_time_start]
    puts "Time after RUN command : [clock format $sys_time_end
        -format %H:%M:%S]"
    puts "Sample download time : $sys_time_difference seconds"

    puts "Downloading of samples to host computer is complete."
    puts "Writing VCD format file of sampled data."
    write vcd -iice iice_dtd -comment {Instrumentor-created
        VCD dump} -gtkwave -noequiv debug$i.vcd

    incr i

    puts "Continue Y/N"
    set keystroke [gets stdin]
    puts $keystroke
}
```

Instrumenting the Design After Compiling

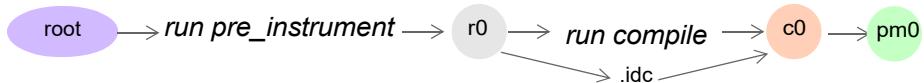
Typically, you instrument a design before it is compiled, as described in [Instrumenting the Design Before Compiling, on page 642](#). The following procedure is an overview of post-compile instrumentation, when you start with a compiled netlist rather than the RTL source files.

This approach offers slightly less visibility into the design than pre-compile instrumentation, but the design is at a more stable stage, with the RTL elaborated. You can use complex trigger and muxing options for the instrumentation. The downside to post-compile instrumentation is that some compiler

optimizations might affect observability, and affect mapping to RTL. You could create a script to check post-compile signals against the RTL to instrument the design and flag mismatches.

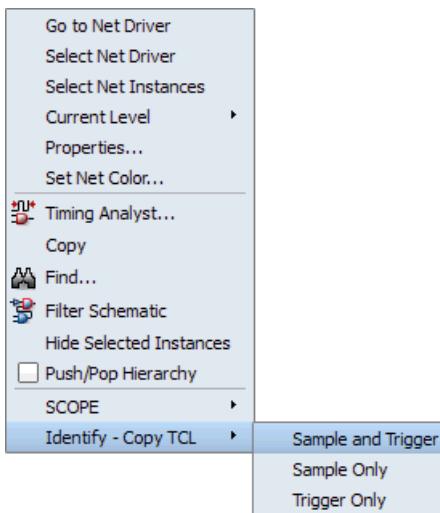
When working with a compiled netlist, you instrument the signals directly in the compiled netlist file, outside the instrumentor. This allows you to update instrumentation that was inserted previously. It also allows you to instrument signals within a parameterized module, which were unavailable for instrumentation before compilation.

1. Instrument a design and compile it with run compile.



You must use `run pre_instrument` first before instrumenting the compiled database.

2. From the compiled database state (`c0`), run the `view schematic` command to open a schematic of the compiled database.
3. Instrument the signals you want from the schematic.
 - Select the signal you want to instrument or update.
 - Right-click the signal, and set the type of instrumentation you want by selecting `Identify - Copy TCL` from the popup menu, and selecting the kind of sample or trigger instrumentation you want to use.



For some guidelines on instrumenting the design and debug planning, see [Recommendations for Debug Planning, on page 647](#).

4. Add the instrumented signal to the idc file.

- Paste the signal string into the idc file. You can create a new idc file or update an existing one. If you are creating a new file, you must add the IICE definition shown on lines 1-3 in the figure below (iice new, iice controller and iice sampler commands for defining a new IICE, configuring the controller, and setting IICE sampler options respectively).

The figure shows the `ctrl_ack_o_0_sqmuxa` signal (from the `block_xfer` block) on line 10 pasted into the idc file as a sample-only signal.

```

1 iice new {IICE} -type regular
2 iice controller -iice {IICE} none
3 iice sampler -iice {IICE} -depth 128
4
5 signals add -iice {IICE} -silent -trigger {/SRS/blk_xfer_inst/cntrl_ack_o_0_sqmuxa}
6 signals add -iice {IICE} -silent -sample {/beh/arb_inst/reset}
7 signals add -iice {IICE} -silent -trigger -sample {/beh/arb_inst/beh/curr_state}\ 
8 {/beh/arb_inst/beh/next_state}
9 iice clock iice {IICE} -edge positive {/clk}
10 signals add -sample {/SRS/blk_xfer_inst/cntrl_ack_o_0_sqmuxa}
```

- Edit the entry and add an -iice option to the line as the following example shows. The Copy TCL command does not automatically include the IICE unit in the entry, so you must add it manually:

```
signals add -iice {IICE} -sample
{/SRS/blk_xfer_inst/cntrl_ack_o_0_sqmuxa}
```

- Save the edited idc file.

5. Run pre-map.

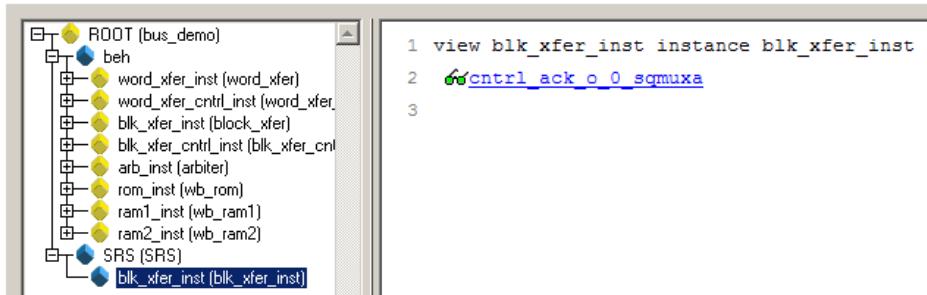
```
run pre_map -idc my_instr.idc premap_args
```

6. Continue with the implementation flow as usual to map, place, and route the design.

7. Set up the design and start the debugger.

For details, see [Running the RTL Debugger, on page 681](#).

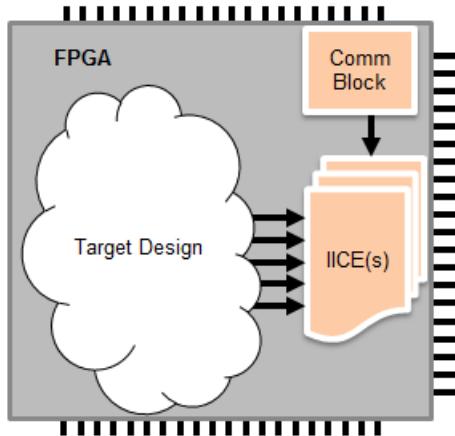
- Run the export runtime command to export the requisite files and set up the design for the debugger.
- Start the debugger with the protocompiler_runtime commands. When you open the debugger, it includes an SRS entry in the browser on the left. Selecting this entry displays the additional signals that were added to the idc file on the right.



You can select a signal in the instrumentation window to open the Watchpoint Setup dialog box, where you can assign a trigger expression to the defined signal. Trigger expressions on signals that are added to the idc file must use the VHDL format.

Defining IICE Parameters

IICE™ (Intelligent In-Circuit Emulator) units are blocks for probe and communication logic to trigger and sample the design. You can add multiple IICES to handle multiple clock domains.



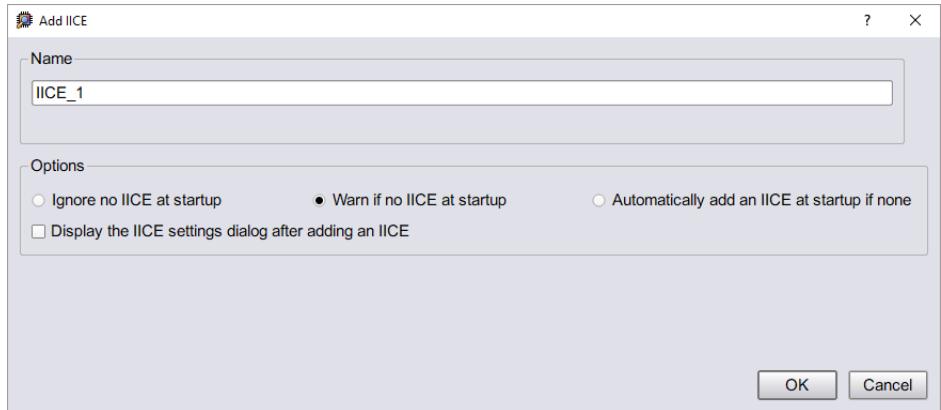
For details, see the following topics:

- [Adding and Deleting IICE Units](#), on page 652
- [Setting Common IICE Parameters](#), on page 654
- [Setting Individual Parameters for Each IICE](#), on page 655
- [Selecting Buffer Type](#), on page 657
- [Specifying IICE Clocks](#), on page 659

Adding and Deleting IICE Units

You can add IICES from the GUI or from the command line.

1. To add an IICE, use either the command line or the GUI as described below:
 - From the command line, use the `iice new` command. For the syntax details, see [iice](#), on page 70 in the *HAPS ProtoCompiler Debugging Environment Reference*.
`iice new [iiceID] [-type rtd|regular]`
 - From the instrumentor GUI, select the Add IICE icon



- Set the type of IICE to regular, unless you are setting up for real-time debugging.
- Optionally, specify a name for the IICE. The default name starts with IICE0 and increments sequentially, but you can specify any name.

When you add an IICE, the compile schematic shows the IICE in the status panel on the left.

2. Select Instrumentor->Instrumentor Preferences and set common parameters that affect all the IICES in the design.

See [Setting Common IICE Parameters, on page 654](#) for details.

These parameters are common to all the IICES in the design; for example, the communication port setting.

- From the command line, use the device command. From the GUI, select Instrumentor->Instrumentor Preferences, and set the options in the Instrumentor Preferences dialog box.
- 3. Set parameters for each individual IICE.

See [Setting Common IICE Parameters, on page 654](#), [Setting Individual Parameters for Each IICE, on page 655](#), [Selecting Buffer Type, on page 657](#), and [Specifying IICE Clocks, on page 659](#).

4. To delete an IICE, use one of these methods:
 - From the instrumentor GUI, first select the IICE and then select Instrumentor->IICE >Delete IICE.

- To delete the IICE from the command line, use `iice delete iiceID`.

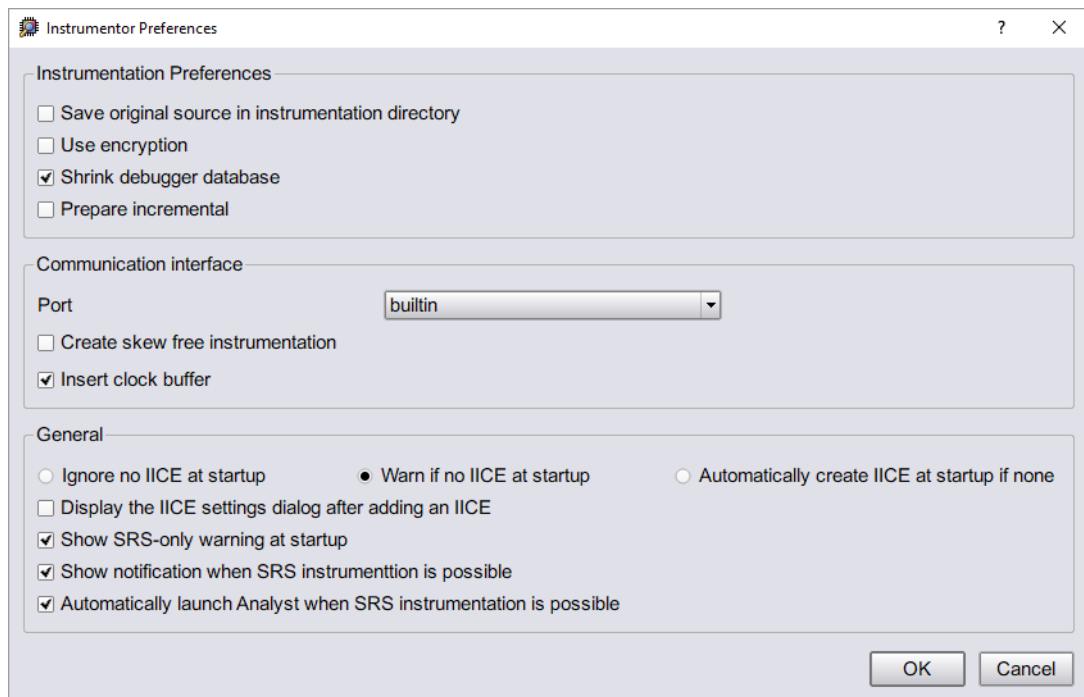


If the IICE configuration parameters for the active IICE need to be changed, use the Edit IICE icon to change them. [Chapter 2, IICE Configuration](#), discusses how to set these parameters for both single- and multi-IICE configurations and for the HAPS deep trace debug feature.

Setting Common IICE Parameters

The common parameter settings apply to all the IICES in the design. They relate to design file settings and communications.

1. Select Instrumentor->Instrumentor Preferences from the HAPS ProtoCompiler menu, or click Communication Interface in the control panel for debug point insertion.



2. To include the original source files, enable the Save original source in instrumentation directory check box.

Use this option to simplify design transfer when debugging for example, is performed on a separate machine. It is especially useful when a design is debugged on a system that does not have access to the original sources. To include the original HDL source with the exported design files, follow these steps:

3. To encrypt the original source files, do the following:
 - Enable the Use encryption check box. When enabled, the original source files are encrypted when they are written, so debug can be run on a machine that otherwise might not be sufficiently secure.
 - Specify a password when prompted. Note that the password is your responsibility, and Synopsys cannot recover a lost or forgotten password. Use these guidelines to set a secure password:
 - Make passwords greater than 16 characters in length.
 - Include numbers, punctuation marks, and spaces.
 - Use spaces to create phrases of four or more words (multiple words defeat dictionary-type matching).
4. Set criteria for the communication interface.
5. Select File->Save from the main HAPS ProtoCompiler menu to save the debug points you set up.

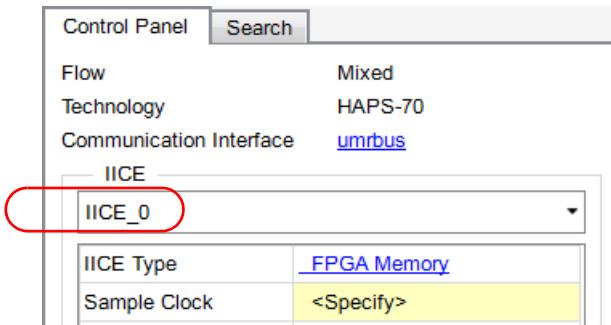
This generates an idc file and adds compiler constraints to the design HDL for the tagged signals and break points. This information is used during the synthesis stages to incorporate the IICE and COMM blocks for the debug logic into the synthesized netlist.

You can now define parameters individually for each IICE as described in [Setting Individual Parameters for Each IICE, on page 655](#).

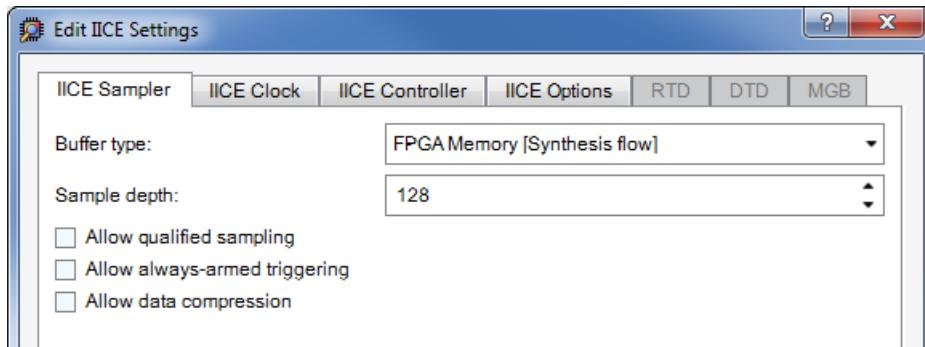
Setting Individual Parameters for Each IICE

In addition to the general settings that apply to all IICE units (see [Setting Individual Parameters for Each IICE, on page 655](#)), you can customize parameter settings for each IICE.

1. Select the target IICE in the Control Panel tab of the debug point insertion window.



2. Open the Edit IICE Settings dialog box by clicking the Edit IICE icon () in the debug point insertion window, or by clicking the entry for IICE Type in the Control Panel.



3. On the IICE Sampler tab, set these options:
 - For details about buffer memory, see [Selecting Buffer Type, on page 657](#).
 - Set the sample depth according to the amount of FPGA RAM available. If you are using deep trace debug, you can set a larger sample depth. The minimum value is 8.
4. Define the sample clock and the clock edge when the samples are taken on the IICE Clock tab, as described in [Specifying a Sample Clock, on page 659](#).
5. Set triggering options.

- Set trigger logic settings for the IICE on the IICE Controller tab.
 - Set additional parameters for importing external trigger signals and exporting trigger signals from the IICE on the IICE Options tab.
6. Set options specific to the kind of debug you are going to run later.
- If you are using real-time debug, set options on the RTD tab. The IICE type must be real-time debug to make these options available.
 - If you are using deep trace debug (DTD), set options on the DTD tab. To see these options, the buffer type must be set to Daughter Board DTD on the IICE Sampler tab.
 - If you are using multi-FPGA DTD, set options on the MGB tab. To see these options, the buffer type must be set to Multi-FPGA DTD Module on the IICE Sampler tab.

Selecting Buffer Type

The buffer type specifies the type of memory used to capture the on-chip signal data for debug. The type of memory you select depends on the hardware and your design needs.

To use an external logic analyzer for debug, set up real-time debug with a new IICE (`iice new` command) instead of specifying a buffer type. For details about real-time debugging, see [Using Real-time Debugging, on page 744](#).

1. You can set the buffer type from the GUI or the command line:
 - Start the instrumentor.
 - To set the buffer type from the GUI, select the IICE icon, and set the buffer type from the IICE Sampler tab.
 - To use the command line, include this command in the idc file:

```
iice sampler -iice {iiceID | all} bufferType
```
2. Specify the type of buffer memory:

| Buffer Type | Command | Supported Hardware Platform |
|-------------------------------------|---|--|
| BRAM (Built-in memory) | <code>iice sampler -iice name internal_memory</code> | All |
| DDR3 (DDR3 memory) | <code>iice sampler -iice name haps_dtd</code> | HAPS-DX7: Onboard memory HAPS-70: External daughter board HAPS-80: External daughter board |
| Built-in DDR3 (DDR3 memory) | <code>iice sampler -iice name haps_DTD_builtin</code> | HAPS-80: Onboard memory |
| DDR4 (DDR4 memory) | <code>iice sampler -iice name haps_dtd</code> | HAPS-80: External daughter board |
| HAPS-DX7 debug hub (DDR3 memory) | <code>iice sampler -iice name haps_dtd2</code> | HAPS-70 HAPS-80 |

- BRAM:
Instrumentor logic that uses distributed RAM blocks (part of the FPGA resources) to store sample data. You can use this for single-FPGA or multi-FPGA debug. To use BRAM for multi-FPGA debug, specify the buffer type in an .idc file that is input at the top level, either at the top-level compile state (RTL-based instrumentation) or the pre-partition state (SRS-based instrumentation). To use BRAM for single-FPGA debug, specify it in an .idc file as an input during the synthesis flow. See [Using BRAM for Debugging, on page 743](#) for details.
- DDR3 Daughter board:
Instrumentor logic that uses external DDR3 memory to store sample data. Allows for larger memory depth as compared with BRAM based instrumentation.
- On HAPS-70 and HAPS-80 systems, use external DDR3 memory to debug a single partitioned FPGA in a multi-FPGA design or to run debug on a single FPGA in multi-design mode. See [Running Deep Trace Debug \(DTD\), on page 702](#).
- HAPS-80 built-in DDR3:
Instrumentor logic that uses built-in DDR3 memory on FPGA A of HAPS-80 systems. This is only available for single FPGA debug. For details, see [Running Single-FPGA Debug on a HAPS-80 FPGA, on page 703](#).
- DDR4 Daughter board:
Instrumentor logic that uses an external DDR4 memory card to store sample data. Allows for larger memory depth as compared with

BRAM based instrumentation. This is only available for HAPS-80 partitioned single FPGAs. For more information, see [Running Single-FPGA Debug on a HAPS-80 FPGA, on page 703](#).

- Off-chip HAPS-DX7 memory module:

Instrumentor logic that uses the an external HAPS-DX7 system as a debug hub to amalgamate data across multiple instrumented FPGAs. Sample data is stored on a DDR3 memory module on the HAPS-DX7 system. Use this buffer type for partitioned multi-FPGA designs. See [Using a Debug Hub with SATA \(DTD2\), on page 716](#) for more information.

Specifying IICE Clocks

You must define a sample clock for each IICE so that the tagged points can be used. In addition, you can use oversampling and define multiple sub-sample clocks for the same IICE. See the following for details:

- [Specifying a Sample Clock, on page 659](#)
- [Defining Multiple Sub-Sample Clocks for Oversampling, on page 660](#)

Specifying a Sample Clock

Define one sample clock per IICE. The sample clock determines when signal data is captured by the IICE, and is required to effectively use the watchpoints and breakpoints that have been tagged. You can specify the sample clock from the GUI or the command line.

1. Determine the signal to use as the sample clock, using the criteria below:
 - The clock signal must be a single-bit scalar signal.
 - The source of the sample clock must be the output of a BUFG or IBUFG component.
 - In general, make sure sampled signals are synchronous with the sample clock or are synchronous with a multiple of the clock, so that the sampled signals are stable when the specified edge of the sampled clock occurs.
2. Mark the signal as the sample clock, using one of these methods:
 - To use the command line, specify the `iice` clock command. Specify the clock edge to be used for triggering with the `-edge` argument. See [iice](#),

[on page 70](#) of the *HAPS ProtoCompiler Debugging Environment Reference* for the complete syntax and options.

- In the GUI, go to the RTL tab that displays the source files, right-click the watchpoint icon on the signal you want to select, and select Sample Clock from the popup menu. The icon changes to a clock face to indicate that it has been specified as a sample clock.

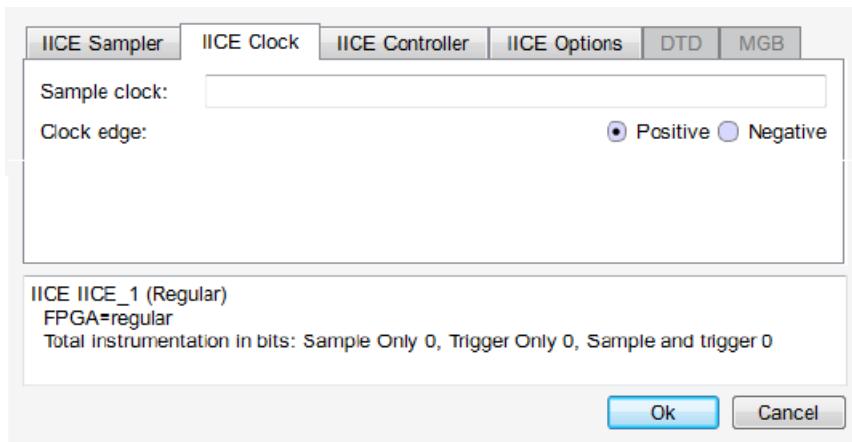
```

41 port (
42    clk           : in
43    reset         : in
44    addr_o        : out
45    block_size    : out

```

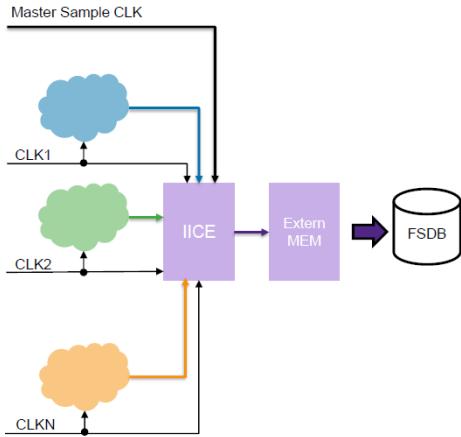
Set the clock edge to be used for triggering from the IICE Clock tab, as described below.

- From the GUI, select the Edit IICE icon () and go to the IICE Clock tab. Specify the sample clock and the clock edge to be used for triggering, and click OK.



Defining Multiple Sub-Sample Clocks for Oversampling

With multi-clock support in one IICE, signals within an IICE can be triggered internally, so cross-triggering is not required. This eliminates the need to use different IICES for asynchronous clocks. Instead, oversample different clocks by adding them to clock groups and synchronizing these clock domains to an oversampling clock domain. Both IICE and memory operate on the sample clock domain. There is one debug controller per design and a single waveform view.



1. Specify the sample clock.

Use the iice clock command or one of the other methods described in [Specifying a Sample Clock, on page 659](#). The sample clock is used as the fast sample clock for oversampling. The sample clock frequency must be at least as fast as the fastest user-defined sample clock domain.

2. In the IICE file, specify the clock signal for the user-defined clock domain with the `clock_group` command.
 - These clocks must be within the same IICE.
 - The sub-sample clocks must be the same speed or lower in frequency than the fast sample clock.
 - The sub-sample clocks can be asynchronous to the fast sample clock
 - Specify the `clock_group` command:

```
clock_group add -name value -freq value -edge value [signal] [-iice value]
```

For command syntax details, see [clock_group, on page 37](#) in the *Debugging Environment Reference Manual*.

Signals that are not defined as part of a clock group are sampled in the fast clock domain. The signals that are defined are sampled at the speed of the clocks defined for that domain.

3. When tagging signals for sampling and triggering, do the following:
 - Include the clock group name:

```
signals add [signal] [-iice value] [-sample] [-trigger] [-msb value] [-lsb value]
           [-field value] [-silent] [-clkgroup value]
```

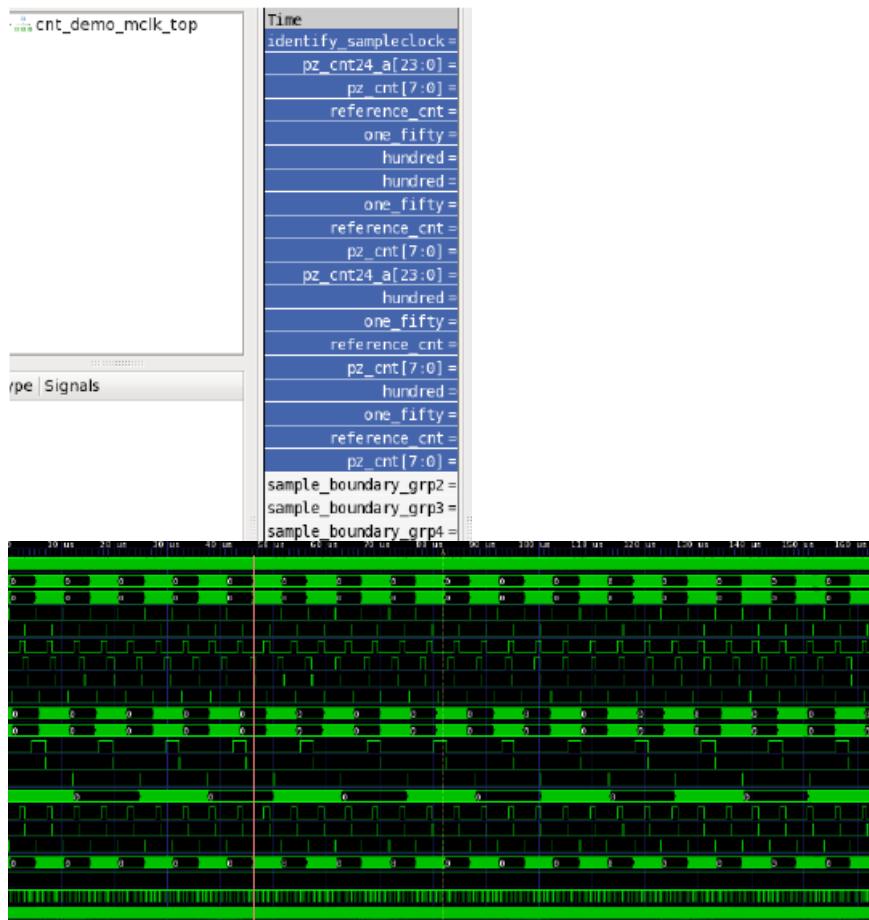
- in the *Debugging Environment Reference Manual* When selecting watchpoints, select them within one clock group, not across different clock groups. This is because watchpoints in a particular clock group are always aligned to the trigger machine but not across clock groups.

The tool automatically creates FIFOs to buffer data captured from asynchronous clock domains, so that they can be displayed by the fast sample clock.

4. View the waveform.

The minimum resolution for the displayed waveforms is the fast sample clock period, but the sub-sample clocks could be asynchronous to the fast sample clock.

The figure shows sample data for signals in different clock groups. The `sample_boundary*` signals specify when the sample data is valid.



You can select watchpoints only within a clock group. Selecting watchpoints across clock groups are not allowed because watchpoints to a particular clock group are always aligned to the trigger machine and not aligned across clock groups.

Making Incremental Instrumentation Changes

If you need to add or remove probe signals during debugging, you typically need to recompile. You can run the compiler incrementally instead of recompiling to reducing runtime.

1. Start with a compiled database with previous instrumentation.
2. Enable incremental debug with this command:

```
set option incremental_debug 1
```

Set this option globally to compile incrementally instead of recompiling the entire design.

3. To make signal changes and run incremental debug from the compile state, use this command:

```
database apply_state -idc | -idclist file [-overwrite]
```

Use this command to specify additional probe signals. With this command, the tool only recompiles modules affected by the IDC changes.

If you specify *-overwrite*, the signals in the file overwrite and replace what was used on the previous run. If you do not specify this parameter, the signals specified in *-idclist* are appended to what was previously used.

| | |
|--|--|
| First compile | <code>database load mydb -autocreate run compile -srclist src.txt -idc my_idc.idc -out c0</code> |
| Incremental compile: Replaces instrumentation from the previous run, <i>my_idc.idc</i> | <code>database load mydb set_state {c0} database apply_state -idc my_new_idc.idc -overwrite</code> |
| Incremental compile: Appends instrumentation from <i>my_new_idc.idc</i> to <i>my_idc.idc</i> | <code>database load mydb set_state {c0} database apply_state -idc my_idc_1.idc</code> |

4. Recompile the design.

The tool runs incrementally, only recompiling modules that are affected by the instrumentation changes.

Example

```
# First compile
database load mydb -autocreate
run compile -srclist src.txt -idc my_idc.idc -out c0
```

```
# Incremental compile
# Replaces (overwrites) previous instrumentation in my_idc.idc
database load mydb
set_state {c0}
database apply_state -idc my_idc_1.idc -overwrite

# Incremental compile
# Adds new instrumentation to existing; does not overwrite
database load mydb
set_state {c0}
database apply_state -idc my_idc_1.idc
```

Making Incremental Changes in UC Instrumentation

Incremental debug lets you modify the instrumentation signals and rerun, without having to recompile the entire design. Incremental debug is a useful way to address these common scenarios:

- To remove instrumented signals in over-instrumentation scenarios.
- To modify the IICE settings and add new instrumentation to an IICE block.
- To change the instrumentation type. By default, all signals specified using \$dumpvars are instrumented as sample and trigger. You can change the instrumentation to sample only or trigger only.
- To add clock groups and mux groups for multi clock and mux set features.

The following procedure provides details on running incremental debug on RTL instrumentation:

1. Set this option to enable incremental debug: option set incremental_debug 1
You can view the results at different stages of the design using the IDC file and debug visibility reports.
2. From the compile stage, export the IDC and modify the IICE settings as needed.
You can only make instrumental modifications at the compile stage for incremental debug.
You can adjust the sample depth and trigger type, or add and delete signals. but you cannot change the IICE name. Note the following commands and syntax details:

- To open the idc file, run the `export idc -path value` command to export the IDC.

- In the file, delete instrumented signals with the `signals delete` command. For example:

```
signals delete -iice {IICE_UC} {top.inst1.sig1}
```

- To instrument new signals, use `signals add`: For example:

```
signals add -iice {IICE_UC} {top.inst1.temp}
```

- Hierarchical signal paths must start with the top-level module name.
- A dot (.) is used as the signal path hierarchy separator, not the slash (/).

- Signals in the generate scope must be specified with square brackets [] instead of parentheses (), because there is no need to specify the range for a signal in IDC.

- VHDL signal paths do not need an architecture name. For example:

```
iice clock -iice {IICE_UC} -edge positive {top.clock_c}
signals add -iice {IICE_UC} -silent -trigger -sample
{top.i1.i2.s1}
```

3. To use the debug visibility report, do the following:

- For single-FPGA designs, type this command from a compiled or pre-map database state. For multi-FPGA designs, run this command from the system-level compile, pre-partition, and system generate stages, as well as the FPGA-level pre-map stage.

```
report debug_visibility -idc <modified.idc> -export_path path
```

The debug visibility report provides a comprehensive report for the signals. It shows the required signals and the stage where they are required, and reports whether the signal is available at the required stage.

```
***** Debug Visibility Report *****

Required signal: count_top.count_top_1.assert_signal
Not available: count_top.count_top_1.assert_signal
Signal not part of the database. Possible cause wrong RTL path.

*****
Required signal: SRS.count_top_14.count[18]
Available at: pre_partition
Available as: count_top_14.count[31:0]

*****
Required signal: SRS.count_top_10.count[28:16]
Available at: pre_partition
Available as: count_top_10.count[31:0]

*****
Required signal: count_top.clk
Available at: pre_partition
Available as: clk

*****
Required signal: count_top.count_top_1.assert_signal
Available at: pre_partition
Available as: count_top_1.assert_signal
```

The command also generates an IDC file. At the compile, pre-partition, and pre-map stages, this IDC is based on the compiled database, with RTL signal paths.

```

hierarchy delimiter .
device jtagport unibus
device prepare_incremental 1
lIce new {IICE UC} -type regular
lIce controller -lIce {IICE UC} none
lIce sampler -lIce {IICE UC} -depth 1024
lIce clock -lIce {IICE UC} -edge positive {SRS.clk}
signals add -lIce {IICE UC} -silent -sample -trigger {SRS.count_top_18.count[31:0]} \
{SRS.count_top_18.assert_signal} \
{SRS.count_top_7.assert_signal} \
{SRS.count_top_5.assert_signal} \
{SRS.count_top_9.assert_signal} \
{SRS.count_top_10.assert_signal} \
{SRS.count_top_16.assert_signal} \

```

When run from the system generate stage, it generates a separate IDC file for each FPGA with the signals that can be instrumented for that FPGA. The following figure shows IDC files for FPGAs A and B from this stage.

```

hierarchy delimiter .
device jtagport unibus
device prepare_incremental 1
lIce new {IICE UC} -type regular
lIce controller -lIce {IICE UC} none
lIce sampler -lIce {IICE UC} -depth 1024
lIce clock -lIce {IICE UC} -edge positive {SRS.clk}
signals add -lIce {IICE UC} -silent -sample -trigger {SRS.count_top_14.
count[31:0]} \
{SRS.count_top_18.count[31:0]} \
{SRS.count_top_12.assert_signal} \
{SRS.count_top_12.assert_signal} \
{SRS.count_top_14.assert_signal} \
{SRS.count_top_15.assert_signal} \
{SRS.count_top_17.assert_signal} \
{SRS.count_top_18.assert_signal} \

```

Generated IDC for FPGA A

```

hierarchy delimiter .
device jtagport unibus
device prepare_incremental 1
lIce new {IICE UC} -type regular
lIce controller -lIce {IICE UC} none
lIce sampler -lIce {IICE UC} -depth 1024
lIce clock -lIce {IICE UC} -edge positive {SRS.clk}
signals add -lIce {IICE UC} -silent -sample -trigger
{SRS.count_top_16.assert_signal} \
{SRS.count_top_1.assert_signal} \
{SRS.count_top_3.assert_signal} \
{SRS.count_top_5.assert_signal}
```

Generated IDC for FPGA B

- Modify the generated IDC as described in step 2 and then apply the modified IDC as described in step 4.
 - For partitioned designs, edit the IDC files for an individual FPGA and then apply the modified IDC to the pre-map stage of that FPGA.
4. To incrementally update the design return to the compile stage and use the modified IDC to recompile the design:

```
database apply_state -idc <modified_idc.idc>
```

This is an example of the sequence of commands:

```

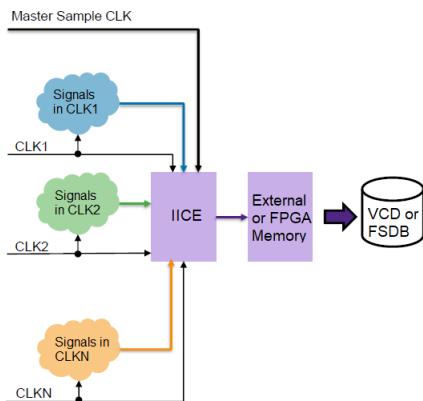
option set incremental_debug 1
launch uc -utf run.utf -ucdb ucdb
run compile -ucdb ucdb -idc my_idc.idc
export idc -path unified_debug_idcs
database apply_state [-idc <modified_idc>] [-idclist
modified_idc_list_file_path]
```

5. To run incremental debug with post-compile instrumentation, do the following:
 - Run compile.
 - Edit the instrumentation in the generated IDC file.
 - Specify the updated IDC file when you run pre-partition or pre-map stages of the flow. Make sure to specify the correct IDC file. If you use a previous file with RTL instrumentation, you get an error.

Instrumenting Multiple Clocks in a Single IICE

Traditionally, separate Intelligent In-Circuit Emulators (IICE™) were defined for each clock domain, and the associated were instrumented and debugged in their respective IICES. With multi-clock instrumentation, you can instrument multiple clock domain signals in a single IICE. To do this, you group signals in different clock groups.

The following block diagram provides a top-level overview of the multi-clock feature. The single IICE contains multiple clocks. Signals associated with different clock domains within the IICE can be triggered. This eliminates the need to use different IICES for asynchronous clocks. This means that cross-triggering is not required and the instrumented signals from different clock groups can be viewed in a single waveform viewer session.



The following steps describe how to instrument multiple clocks in one IICE:

1. Create the database and pre-instrument the design.
2. Use FDC constraints to define all the clocks in the design.
 - Define clocks with the `create_clock` command. If there are clocks derived from the design, define them in FDC using `create_generated_clock` command.
 - Group the clocks as needed using the `set_clock_groups` command. The FDC file for the example shows the top-level asynchronous clocks grouped in the FDC file:

```
create_clock -name {clk_40} {p:clk_40} -period {25}
create_clock -name {clk_30} {p:clk_30} -period {33.33}
create_clock -name {clk_20} {p:clk_20} -period {50}
create_clock -name {clk_10} {p:clk_10} -period {100}

set_clock_groups -asynchronous -name clk40_grp1 -group {clk_40}
set_clock_groups -asynchronous -name clk30_grp2 -group {clk_30}
set_clock_groups -asynchronous -name clk20_grp3 -group {clk_20}
set_clock_groups -asynchronous -name clk10_grp4 -group {clk_10}
```

3. Open the instrumentor, and create a new IICE.
4. Specify the fastest clock in the design as the sample clock for that IICE. The sample clock (or fast clock) must always be active. Sample clocking must not be stopped once it is started. The sample clock must not be a gated clock. The sample clock must not be defined in clock group, as this is the fastest clock in the design.
5. Group multiple clocks using the `clock_group` command.

Clocks in the same group are synchronous, so use different clock groups for asynchronous clocks. Once clock groups are defined for each clock domain, the signals can be instrumented in these clock groups. For example, if `sig_a` and `sig_b` signals are from two different clock domains, `clk_a` and `clk_b`, then the signals are instrumented as follows:

```
clock_group add -name group1 -freq {30} -edge {pos} {/clk_a} -iice {IICE_0}
clock_group add -name group2 -freq {50} -edge {pos} {/clk_b} -iice {IICE_0}
signals add -iice {IICE_0} -clkgroup {group1} -silent -sample -trigger {/sig_a}
signals add -iice {IICE_0} -clkgroup {group1} [-silent] -sample [-trigger] {/sig_a}
```

6. In the IDC file, instrument the signals in different clock domains by grouping them together in different clock groups.

A sample of the IDC file is shown below:

```

device jtagport umrbus
device umr_pipe 1
device prepare_incremental 1
iice new {IICE_0} -type regular
iice controller -iice {IICE_0} none
iice sampler -iice {IICE_0} -depth 2048
iice sampler -iice {IICE_0} -pipe 3

iice clock -iice {IICE_0} -edge positive {clk_40}
signals add -iice {IICE_0} -silent -trigger -sample {/cnt_64}
signals add -iice {IICE_0} -silent -trigger -sample {/valid_40}

clock_group add -iice {IICE_0} -name grp2_30 -freq {30} -edge {POS} {/clk_30}
clock_group add -iice {IICE_0} -name grp3_20 -freq {20} -edge {POS} {/clk_20}
clock_group add -iice {IICE_0} -name grp4_10 -freq {10} -edge {POS} {/clk_10}

signals add -clkgroup grp2_30 -iice {IICE_0} -silent -trigger -sample {/cnt_128}
signals add -clkgroup grp3_20 -iice {IICE_0} -silent -trigger -sample {/cnt_192}
signals add -clkgroup grp4_10 -iice {IICE_0} -silent -trigger -sample {/cnt_256}

signals add -clkgroup grp2_30 -iice {IICE_0} -silent -trigger -sample {/valid_30}
signals add -clkgroup grp3_20 -iice {IICE_0} -silent -trigger -sample {/valid_20}
signals add -clkgroup grp4_10 -iice {IICE_0} -silent -trigger -sample {/valid_10}

```

- Continue with the normal design flow from compile to bit-file generation of individual FPGAs, and export the design for runtime debugging.

You are now ready for the debug stage of the design, described in [Debugging a Multi-Clock Design, on page 684](#).

Capturing Startup Errors with Pre-Armed Triggers

You can pre-arm triggers to catch errors from the configuration state, that you would otherwise not catch. Preconfiguring or pre-arming triggers lets you capture error events that occur immediately after reset during startup or initialization, before the system is operational. The typical debug flow does not cover this window, because you normally configure triggers after the initial startup. You can use preconfigured triggers to debug reset initialization, clock tree issues, hardware locking and floating pin defects, among other scenarios.

Pre-arming also lets the prototyping engineer configure the triggers before handing off to software development. When a trigger event occurs, the prototyping engineer can use a push mechanism like email to be notified of a trigger event, or extract the status from the debugger or Confpro.

- In the instrumentor, set watchpoints and breakpoints as usual.
- Use the following commands to specify the conditions that enable preconfigured triggers.

```
breakpoints preconfigure  
signals preconfigure
```

```
statemachine addtrans
```

For the command syntax, see [Chapter 2, Instrumentor and Debugger Commands](#) in the *Debugging Environment Reference Manual*.

In the following example, the first command instruments the signal for the trigger. The second command sets a watchpoint condition on `{/arb_inst/curr_state}` that triggers when the signal transitions from `2'b00` to `2'b01`.

```
signals add -iice {IICE_0} -silent -trigger -sample  
      {/arb_inst/curr_state}  
signals preconfigure {/arb_inst/curr_state} {2'b00} {2'b10} -iice  
      {IICE_0} -condition 0
```

3. Save the idc file.
4. To view results from pre-armed triggers, run the debugger (runtime executable) with this command:

```
run -skip_config
```

This command skips the normal configuration step and downloads sample data from the pre-configured trigger.

If you use the GUI instead of the command line and click Run in the debugger, a dialog box asks you if you want to skip configuration. Click Yes to download sample data from the pre-configured trigger. If you click No, the debugger runs and reconfigures the triggers with the conditions you specify.

Instrumenting with Unified Compile

There are two ways to designate probe signals and instrument the RTL design with the UC flow: using SystemVerilog assertions and using \$dumpvars. In addition to these RTL instrumentation techniques, you can add post-compile instrumentation.

The instrumented signals are managed through the Instrumentation-Debug Constraints file (IDC), which contains constraints for the instrumented signals and break points, as well as the Intelligent In-Circuit Emulator (IICE) which is the debug IP.

See the following for more information:

Instrumentation

- [The UC Instrumentation Flow](#), on page 672
- [Adding UC Instrumentation](#), on page 46 (IG)
- [Setting Instrumentation Limits for UC Designs](#), on page 48 (IG)
- [Specifying Post-Compile Instrumentation for UC Designs](#), on page 48 (IG)
- [Adding SVAs for Instrumentation](#), on page 53 (IG)
- [Detecting Assertion Failures with a HAPS System Reset](#), on page 58 (IG)
- [Adding \\$dumpvars for Instrumentation](#), on page 62 (IG)

IG is the *Instrumentation Guide*

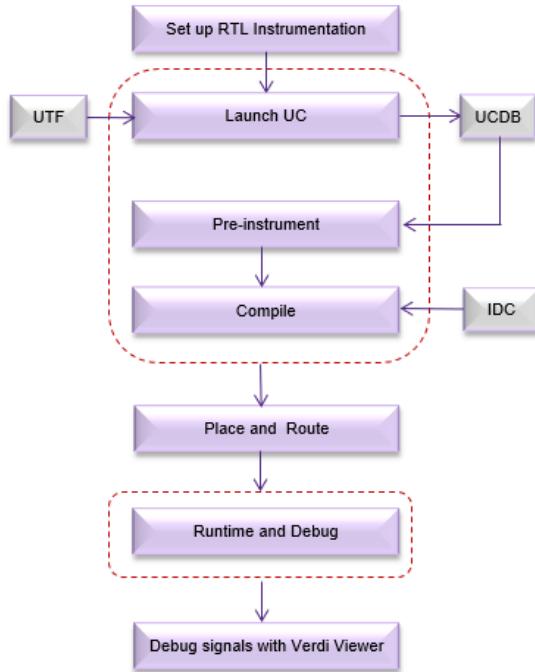
Using IDC and IICE

- [Setting up the IDC and IICE](#), on page 71 (IG)
-

During the runtime phase, you can run the design on the HAPS hardware and debug the design. You can then use the Synopsys Verdi software to check the waveforms from the instrumented signals against a golden version.

The UC Instrumentation Flow

The figure summarizes the steps in the instrumentation flow. The dashed lines indicate the steps that are performed in the prototyping tool.



The procedure below describes the details of the instrumentation flow above. The process applies to both instrumentation modes: SVA or \$dumpvars.

1. Set up the design.
 - Add RTL instrumentation. For detailed flows, refer to the *Instrumentor Guide*. For SVA, see [Adding SVAs for Instrumentation, on page 53](#), and for \$dumpvars, see [Adding \\$dumpvars for Instrumentation, on page 62](#).
 - Run unified compile and generate a compiled database according to the instrumentation mode being used.
 - Run pre-instrument on the compiled database. The run pre_instrument command generates an instrumentation database. The example shows a typical set of commands. The commands are the same for SVA and \$dumpvars except for option set; both are shown in this snippet:

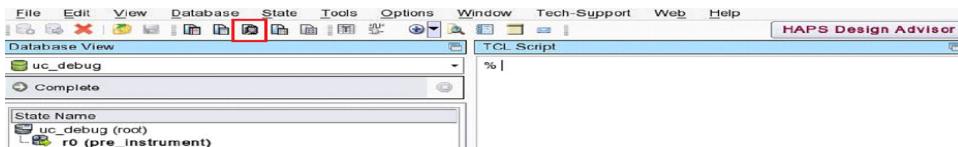
```

database load uc_debug -autocreate -technology HAPS-80
#Option command for SVA
option set debug_sva 1
#Option command for $dumpvars
option set debug_dumpvars 1
launch uc -utf <filename.utf> -ucdb <ucdb_dir_name> -v 2.0
run pre_instrument -ucdb <ucdb_dir_name>

```

2. Instrument the clock settings, using the IDC file.

- In the GUI, select the Edit IDC icon, shown below. The IDC file configures the hardware debug IP, and includes clock specifications. These parameters are required.



- Define required IICE settings like depth, sampling clock, sampling clock edge.
- Instrument the clock.
- Define other functionality as needed:

| Command | Use to... |
|---------------|--|
| iice_new | Create multiple IICE blocks when using different instrumentation modes in the same design, like SVA and \$dumpvars. See <i>iice new</i> in the <i>HAPS® Command Reference Manual</i> . |
| trigger_group | Group multiple SVA nets together to generate a single-bit net. See <i>trigger_group (IDC Command)</i> in the <i>HAPS® Command Reference Manual</i> . |

- Save the IDC file. This is an example of a user IDC:

```

hierarchy delimiter .
device jtagport umrbus
device umr_pipe 1
iice new {IICE_UC} -type regular
iice controller -iice {IICE_UC} none
iice sampler -iice {IICE_UC} -depth 1000
iice sampler -iice {IICE_UC} -pipe 3
iice clock -iice {IICE_UC} -edge positive {top.inst1.clock}

```

3. When you compile the design with run compile, specify the generated IDC file.

```
database load uc_debug -autocreate -technology HAPS-80
#Option command for SVA
option set debug_sva 1
#Option command for $dumpvars
option set debug_dumpvars 1
launch uc -utf <filename(utf)> -ucdb <ucdb_dir_name> -v 2.0
run pre_instrument -ucdb <ucdb_dir_name>
run compile -idc <user_idc_name>.idc
export idc -path <user_defined_dir_name>
```

The signals are instrumented during the compile stage and no further instrumentation is required. The compile stage uses the IDC file settings to instrument the probe signals through SVAs or \$dumpvars, as specified in the RTL.

4. Check that the instrumented signals have been correctly implemented.
 - Use the export idc command, see *export IDC (Tool Command)* in the *HAPS® Command Reference Manual*, to view the signals. This command generates a dumpvars_report.txt report in the exported IDC directory, which lists all the signals and their status (applied/not applied). You can compare these signals with the signals in the HDL code under the unified compiled database.
 - Check the log files. After compile, check the signals in the compile log report, *design_name>identify_idc_generator.log*. After the system generate and pre-map stages, check the signals in the instrumentation log of the log report.
 - For SVA, you can also check for assertion failures by doing a system reset, as described in *Detecting Assertion Failures with a System Reset, on page 544*.
5. For incremental modifications, use the export idc command, see *export IDC (Tool Command)* in the *HAPS® Command Reference Manual*.
6. To specify a post-compile instrumentation file with information generated after the compile stage, follow these guidelines:
 - You can specify a new IDC file in addition to the existing file.
 - In the post-compile file, make sure the hierarchical signal path starts with the top-level module.
 - Use a period (.) not a slash (/), as the path hierarchy separator.

- ISpecify signals in the generate scope with square brackets [] instead of brackets (), because there is no need to specify the range for a signal in IDC.

Example:

```
iice clock -iice {IICE_UC} -edge positive {top.clock_c}
signals add -iice {IICE_UC} -silent -trigger -sample
{top.i1.i2.s1}
```

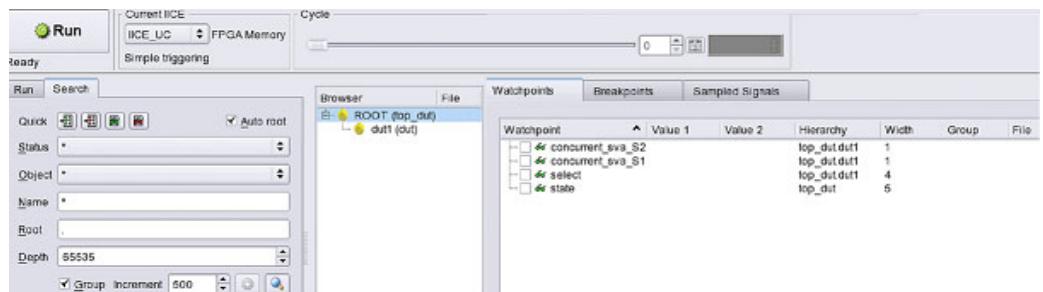
Verifying UC Designs

UC designs are verified by using the Verdi waveform viewer to check results against a golden design. The Verdi software is used for designs with RTL instrumentation as well as non-instrumented designs using global state visibility (GSV). The following sections describe the details:

- [Debugging Designs with SVAs](#), on page 677
- [Running Incremental Unified Debug](#), on page 678
- [Using the Verdi Flow with a UC Design](#), on page 803

Debugging Designs with SVAs

The tool lists all the synthesized assertions with SVA instance names that were passed through the IDC file, along with other debug signals.



1. Enable the debug signal of the SVA instance using regular runtime commands to monitor assertion failures. For example:

```
watch enable -language verilog {top_dut.dut1.concurrent_sva_S2} {1'b0} {1'b1}
```

2. Check for signal value changes in the waveform.

When the assertion fails, the corresponding debug signal value changes from 0 to 1 in the waveform.

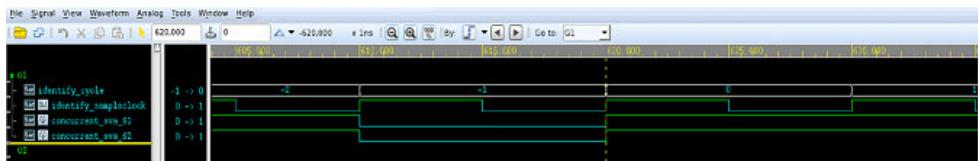
If you evaluate the SVA property with respect to the negedge clock when the IICE sampling is at the posedge clock, or vice versa, the SVA samples appear one cycle earlier in the waveform. For example: assert property (@(negedge clk) req##1 grant).



3. To filter SVAs in the waveform, use these commands:

- `signals set_property -name show_in_waveform -value 0 [hierarchy find -all -maxdepth 100 -name "*" -stat "*" -type "signal" .]`
- `signals set_property -name show_in_waveform -value 1 [hierarchy find -all -maxdepth 100 -name "*" -stat "*" -type "assert" .]`
- write fsdb

The following figure illustrates the filtered SVA signals.



For information on using the Verdi integration flow, see [Using the Verdi Flow with a UC Design, on page 803](#).

Running Incremental Unified Debug

Incremental debug lets you modify the instrumentation signals and rerun, without having to recompile the entire design. Incremental debug is a useful way to address these common scenarios:

- To remove instrumented signals in over-instrumentation scenarios.
- To modify the IICE settings and add new instrumentation to an IICE block.
- To change the instrumentation type. By default, all signals specified using \$dumpvars are instrumented as sample and trigger. You can change the instrumentation to sample only and/or trigger only
- To add clock groups and mux groups for multi clock and mux set features.

The following procedure provides details on running incremental debug on RTL instrumentation:

1. Set this option to enable incremental debug: option set incremental_debug 1.
2. Compile the design, and then run the `export idc -path value` command to export the IDC.

You can only make instrumental modifications at the compile stage for incremental debug.

3. Modify the IICE settings of the exported IDC as needed.

You can adjust the sample depth, trigger type, etc. but you cannot change the IICE name. Note the following commands and syntax details:

- Delete instrumented signals with the `signals delete` command. For example:

```
signals delete -iice {IICE_UC} {top.inst1.sig1}
```

- To instrument new signals, use `signals add`: For example:

```
signals add -iice {IICE_UC} {top.inst1.temp}
```

- Hierarchical signal paths must start with the top-level module name.
- A dot (.) is used as the signal path hierarchy separator, not the slash (/).
- Signals in the generate scope must be specified with square brackets [] instead of parentheses (), because there is no need to specify the range for a signal in IDC.
- Signal path for VHDL does not need an architecture name.

For example:

```
iice clock -iice {IICE_UC} -edge positive {top.clock_c}
signals add -iice {IICE_UC} -silent -trigger -sample
{top.i1.i2.s1}
```

4. To incrementally update the design return to the compile stage and use the modified IDC to recompile the design:

```
database apply_state -idc <modified_idc.idc>
```

This is an example of the sequence of commands:

```
option set incremental_debug 1
launch uc -utf run.utf -ucdb ucdb -v 2.0
run compile -ucdb ucdb -idc my_idc.idc
export idc -path unified_debug_idcs
database apply_state [-idc <modified_idc>] [-idclist
modified_idc_list_file_path]
```

5. To run incremental debug with post-compile instrumentation, do the following:
 - Run compile.
 - Edit the instrumentation in the generated IDC file.
 - Specify the updated IDC file when you run pre-partition or pre-map stages of the flow. Make sure to specify the correct IDC file. If you use a previous file with RTL instrumentation, you get an error.

For information on using verdi integration flow, see [Using the Verdi Flow with a UC Design, on page 803](#).

Running the RTL Debugger

This section provides an overview on how to use the debugger. For more comprehensive information, refer to the help built into the debugger.

- [Launching and Running the Debugger](#), on page 681
- [Varying Buffer Depth Dynamically for DDR3](#), on page 687
- [Viewing Captured Deep Trace Debug Samples](#), on page 689
- [Verifying the Hardware Configuration for Deep Trace Debug](#), on page 690

Also refer to the details about deep trace debug:

- [Working with Deep Trace Debug](#), on page 695
- [Running Deep Trace Debug \(DTD\)](#), on page 702
- [Using a Debug Hub with SATA \(DTD2\)](#), on page 716

Launching and Running the Debugger

1. Validate the hardware setup for the debugger.

Make sure that the hardware is set up properly and will not cause errors. Use the board bring-up utilities to do this. Check for incorrect clock configurations, mismatches with boards not plugged in correctly, and cabling mistakes. Run a live hardware scan to check that the hardware setup and that all connections are functioning correctly.

2. Set up the design for the debugger.

– Verify that the hardware is set up correctly for the debug scheme you are using. See the following for details:

[Verifying the Hardware Configuration for Deep Trace Debug](#), on page 690

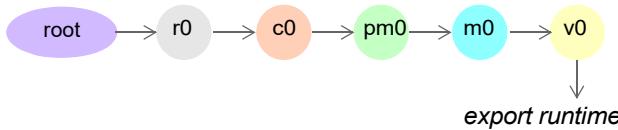
[Working with Deep Trace Debug](#), on page 695

– Check that the runtime package is installed and that the runtime executable is available. This installation is required to run the debugger.

- Instrument a design, and complete synthesis and place-and-route, as described in [Instrumenting the Design for Debug, on page 641](#).
3. Synthesize, place, and route the design, and then export the required design files by typing the following command at the system prompt in the ProtoCompiler tool:

```
export runtime -path path -debug_export_sources
```

The command creates a directory with all the files you need for debugging your design, including the bit file and a project file.

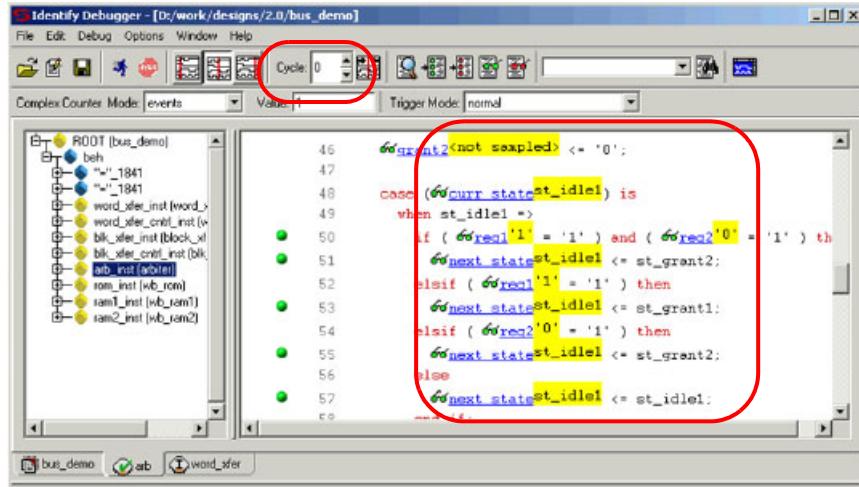


4. Start the debugger by typing the following command at a system prompt:

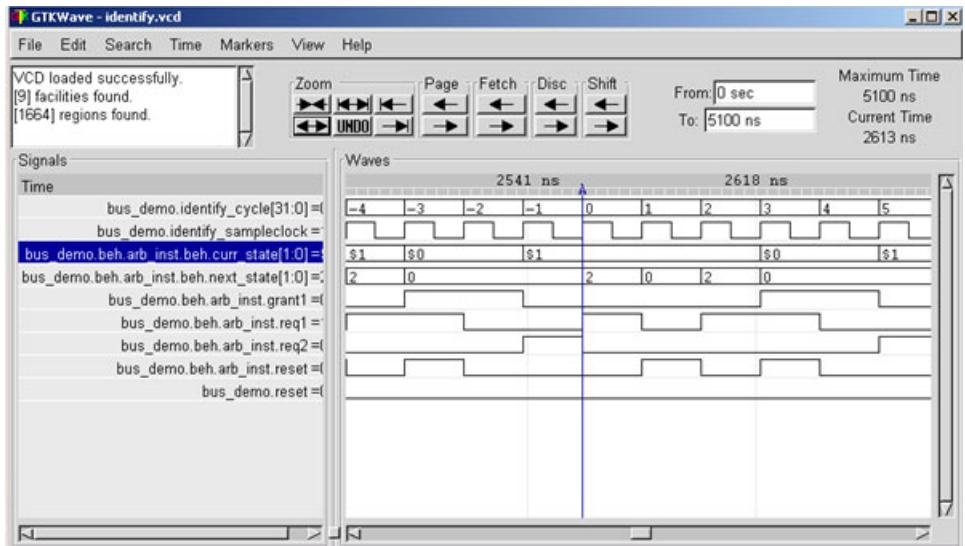
```
protocompiler_runtime debug -in path/debug.prj
```

The debugger window opens.

5. Run the debugger as described in the *Debugger User Guide*.
6. Analyze and debug the FPGA data, based on the instrumentation you inserted earlier.
- In the debugger window, check the source code. The samples are highlighted in yellow. Check the cycle count values in the sample buffer.



- View the waveform data. By default the RTL debugger generates a VCD waveform, which you can view with the GTK waveform viewer. To use other formats and tools, set your preferences in the Default Waveform viewer panel, under Options-> Debugger preferences.



7. For remote debug, set up a connection, with the server on one machine, and the debugger as the client on another.

8. If you have set up the hardware as described in [Ensuring Hardware Bring-up, on page 621](#), you can verify the design on the hardware.

Debugging a Multi-Clock Design

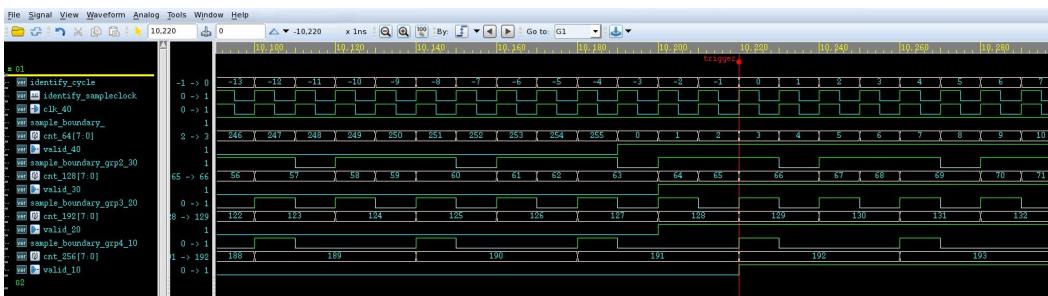
Once you have properly set up the clocks and synthesized the design (see [Instrumenting Multiple Clocks in a Single IICE, on page 668](#)), you can debug the design.

1. Program the hardware using Confpro, defining the frequencies and configuring bit files for each FPGA.
2. Open the debugger, load the project, run link training for DTD, and set the trigger conditions. See [Running Multi-FPGA Debug with HAPS-80 Built-in Memory, on page 712](#). You must do link training each time you modify the configuration settings. Proceeding without link training might result in misaligned waveforms.
3. Run the debugger and observe the captured waveforms.
 - You can view multi-clock domain signal waveforms synchronously.
 - You can view the waveforms in VCD or FSDB format by setting the options in Debugger as follows:

For VCD - Set the waveform type as VCD, Setup Debugger > Waveforms > GTKWave.

For FSDB - Set the waveform type as FSDB, Setup Debugger > Waveforms > Synopsys Verdi.

 - View the sampled signals with reference to their sample_boundaries. See [Sample Boundaries, on page 685](#) for a discussionThe figure below shows the waveforms for sample_boundaries and their associated signals, captured in FSDB format. See [FSDB Waveform Calculations for Clock Groups, on page 686](#) for additional details.



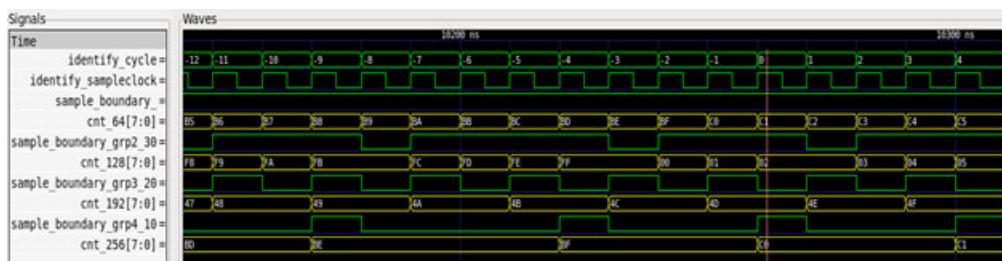
The next figure shows the waveforms for `sample_boundary` and their associated signals, captured in VCD format:



Sample Boundaries

Each clock domain has its own sample boundary. The sample data for signals in different clock groups is captured with reference to the `sample_boundary` for that clock domain.

The minimum resolution for the displayed waveform is the fast sample clock period, but the sub-sample clocks can be asynchronous to the fast sample clock. The waveform example below shows four sample boundary signals (in green) for four different clock domains. Data signals associated with the clock domain (shown in yellow) can be viewed with reference to their sample boundaries.



The `sample_boundary*` signals specify when the sampled data is valid. When the sample boundary is 1 or high, its associated data signal values are valid. And when the sample boundary is 0 or low, the signal values are not valid as the previous data values remain in place.

The software automatically creates FIFOs to buffer data that is captured from asynchronous clock domains, so that they can be displayed by the fast sample clock. The sample clock or fast clock cannot be a gated clock. Note that the sample clock must always be active. Once started, the sample clock must not be stopped.

FSDB Waveform Calculations for Clock Groups

The FSDB waveform depends on the fastest clock frequency, which is the sampling clock frequency for the design. The remaining clock signals in the waveform are shown as de-skewed by n-clock cycles of their period.

The number of clock cycles by which the slower clock signals (n) are de-skewed is calculated using this formula:

$$\text{Number of clock cycles to de-skew}(n) = (\text{Sampling Frequency}/\text{Slow clock frequency}) - 1$$

The figure below shows waveforms for an example design where the sampling clock frequency is 40 MHz, and three additional clocks in the design run at 30 MHz, 20 MHz, and 10 MHz. The signals associated with the 30 MHz clock group (G1) are de-skewed according to the formula below. The 0 indicates that de-skewing is not required for this group.

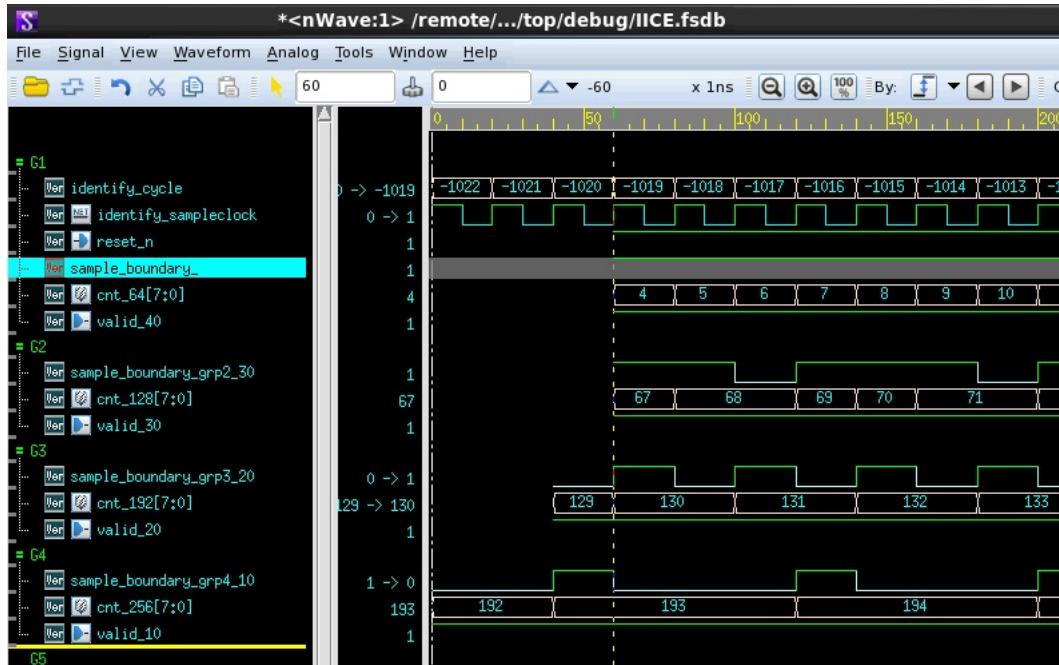
$$\text{Number of clock cycles to de-skew}(n) \text{ for } 30 \text{ MHz clock group} = (40/30) - 1 = 1/3 = 0$$

The waveform shows that the signals in the G2 group are de-skewed by 1 cycle, according to this formula:

$$\text{Number of clock cycles to de-skew}(n) \text{ for } 20 \text{ MHz clock group} = (40/20) - 1 = 1$$

Similarly, the signals in the G3 group are de-skewed by 3 clock cycles in the waveform.

$$\text{Number of clock cycles to de-skew}(n) \text{ for } 10 \text{ MHz clock group} = (40/10) - 1 = 3$$

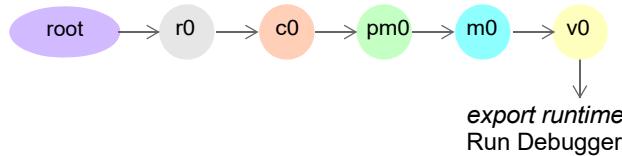


Varying Buffer Depth Dynamically for DDR3

The tool automatically calculates the maximum buffer depth for DTD, based on the number of instrumented bits. See [Running DTD2 with a HAPS-DX7 System and SATA Cabling, on page 718](#). However, you can dynamically vary the buffer depth for DDR3 buffers or DDR3 memory modules. Setting a limit on the buffer depth can help with data offload times.

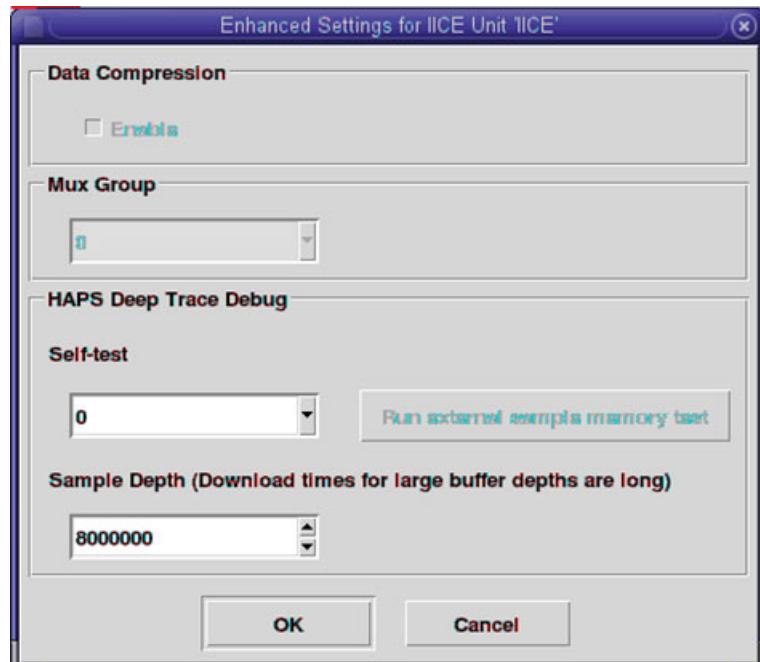
1. Start the debugger with the protocompiler_dx_runtime debug command.

See [Running the RTL Debugger, on page 681](#) for details.



2. In the debugger GUI, click the Enhanced IICE Settings icon () on the right side of the window.
3. Dynamically configure the sample depth in the dialog box that opens.

You cannot set it to a depth greater than the configuration set during instrumentation (see [Running DTD2 with a HAPS-DX7 System and SATA Cabling, on page 718](#)).



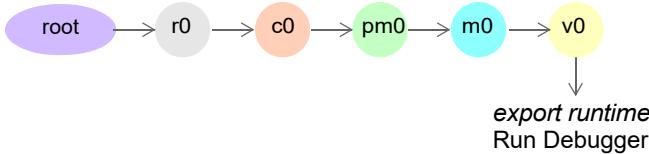
Viewing Captured Deep Trace Debug Samples

If you set a large buffer depth value for DTD, the large sample size can significantly slow down loading time. Captured samples are downloaded block by block. A large sample size translates to large VCD/FSDB files. For these cases, the tool offers you the option of viewing or writing out a slice of the FSD or VCD waveform based the number of captured cycles.

The following steps describe this procedure:

1. Start the debugger with the `protocompiler_runtime` command.

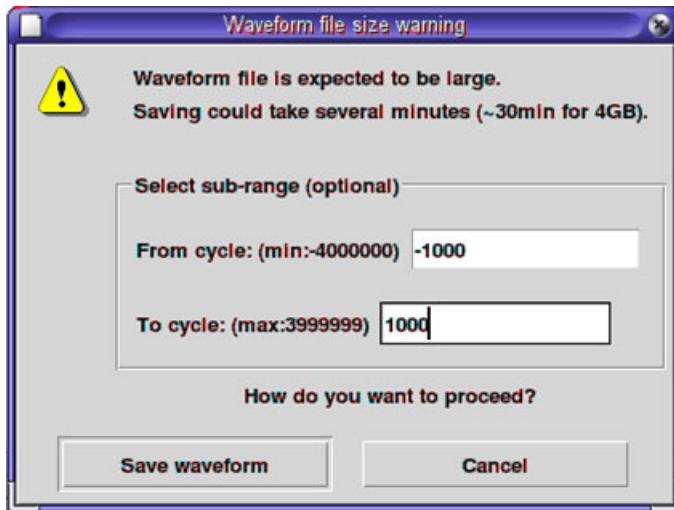
See [Running the RTL Debugger, on page 681](#) for details.



2. In the debugger GUI, click the Waveform Display icon.

If the sample depth is more than 8000000, the tool displays a popup window.

3. Do the following in the pop-up window that opens:
 - Specify the cycle range you want to view on either side of the trigger position. The following example shows a sub-range -1000 to 1000 specified, although the complete VCD/FSDB extends from -4000000 to 3999999 on ether side of the trigger position.



- Click Save waveform at the bottom of the dialog box to save and view the specified sub-range. If you click the button without specifying a sub-range, the tool saves the entire waveform to IICE.vcd or IICE.fsdb. This could take some time, as it downloads the full buffer depth and all the sample blocks. A full VCD dump could take hours.
4. Alternatively, write out vcd or fsdb using the -range argument with the appropriate TCL command:

```
write vcd -range {fromCycle toCycle} filename.vcd
write fsdb -range {fromCycle toCycle} filename.fsdb
```

For example:

```
write fsdb -range {-10 10} test.fsdb
```

Verifying the Hardware Configuration for Deep Trace Debug

You can verify the deep trace debug memory configuration for a HAPS DDR3 daughter board or the Multi-FPGA DTD module by running a self-test.

1. To run the self-test from the debugger console window, use the following command:

```
iice sampler -runselftest 1|0
```

There are two patterns (0 and 1) that you can run. The self-test writes data patterns to the external memory and reads back the data pattern to detect configuration errors and connectivity problems.

2. To run the self-test from the debugger GUI, follow these steps:
 - Open the project view and click the IICE icon.
 - Select pattern 0 or pattern 1 from the Self-test drop-down menu.
 - Click Run external sample memory test button.
3. Repeat the command using the second pattern to ensure adequate testing.
4. You can run the self-test multiple times, at different points in the design cycle:
 - After the initial setup, to verify the hardware configuration against the instrumentation
 - During routine operations, when a hardware problem is suspected
 - When the physical configuration is modified by changing the IICE configuration settings.

Configuring CEL-Based Triggers

Complex Event Language (CEL) is a Verilog-like language to describe state machines. You can use CEL to configure triggers, by using it to describe DUT status monitoring and notification conditions.

1. Create a CEL file.

For information about the file, see [CEL Syntax, on page 27](#) in the *Debugger Reference*.

- For instrumented SRS signals, use the following syntax to add a new hierarchy SRS name before the original hierarchy name. Use the complete SRS signal name defined in instrumentation).

For the standard compile flow:

```
var var_1 top_module.SRS.srs_hierarchy_name;  
var var [msb : lsb] top_module.SRS.\hdl_hierarchy [msb : lsb] ;
```

For the unified compile flow:

```
var var_1 SRS.srs_hierarchy_name;  
var var [msb : lsb] SRS.\hdl_hierarchy [msb : lsb] ;
```

2. From the debugger GUI, select Debugger -> Load CEL.



3. Select the IICE, specify the CEL file, and click Compile.

The equivalent command is cel compile. Other commands are cel set and cel get (see [cel, on page 27](#) in the *Debugger Reference*).

4. Check for errors.

See [CEL Error Reports, on page 692](#) for descriptions of the errors.

CEL Error Reports

There are various kinds of errors that are reported:

- CEL file syntax errors

If there is a syntax error in the CEL file, you get this message: Error: Failed to parse. The error also lists the line where the error occurred and the last token that caused the error. For example:

```
Error: Failed to parse
{syntax error on line 8: last token is event}
```

- Undefined reference errors
 - Variable not found.

These messages begin with VAR_NAME at line NUMBER:. If the variable is missing, use the hierarchy found Tcl command to get a list of available variables.

- Undefined reference in event declarations

These messages begin with Event EVENT_NAME at line NUMBER:. If an undefined reference is found, declare the variable first, followed by a watchpoint activation as described in [CEL Syntax, on page 27](#). It is also essential for an event to be declared first if it is going to be reused.

- State machines with empty operations

These messages begin with State STATE_MACHINE at line NUMBER:. Delete states with empty operations.

- Unsupported operation errors

These error messages indicate the operation is not currently supported. These messages use the following prefixes:

Unsupported operation at line NUMBER:

Unsupported operator at line NUMBER:

- Invalid operation errors

These error messages indicate that the specified usage is not correct. For example, you cannot assign true or false to a multi-bit variable. Most of these messages use the following prefix:

Invalid operation at line NUMBER:

However some invalid operations in variable declarations use a general variable prefix, and the invalid operation error is described in the part after the colon. For example:

VAR_NAME at line NUMBER: Bit-width (X-bit) is not equal to Y-bit in hierarchy name.
VAR_NAME at line NUMBER: Bit-width (X-bit) is not equal to Y-bit in IICE.
VAR_NAME at line NUMBER: Variable range [msbX:lsbX] is not in the range [msbY:lsbY] in IICE.
VAR_NAME at line NUMBER: Cannot do cross triggering with itself.
VAR_NAME at line NUMBER: Cannot assign variable range to identifier/IICE.
VAR_NAME at line NUMBER: Cannot assign variable range to breakpoint.
VAR_NAME at line NUMBER: Signal ESCAPED_NAME is an escaped identifier without backslash.
VAR_NAME at line NUMBER: Signal ESCAPED_NAME is an escaped identifier without space suffix.

- Resource limitation errors

These messages are displayed when the number of conditions or states or other resources are not sufficient. These messages often use this prefix:

Invalid Usage:

However, other resource errors are often only distinguished by the description after the colon. These are some examples of messages with assorted prefixes:

- Counter declaration

Event EVENT_NAME at line NUMBER: Excess maximum counter value MAX_COUNTER_VALUE for occurs, sustains.
At line NUMBER: Excess maximum counter value MAX_COUNTER_VALUE.

- Event declaration

Event EVENT_NAME at line NUMBER: Excess maximum number of MAX_CONDITION_NUMBER conditions.
At line NUMBER: Excess limited operator/operand number MAX_OPERAND_NUMBER.

- State machine declaration

FSM FSM_NAME at line NUMBER: Excess maximum number of states MAX_STATE_NUMBER.
FSM at line NUMBER: Excess maximum number of states MAX_STATE_NUMBER. Introduce new state when using occurs, sustains.

Working with Deep Trace Debug

Deep Trace Debug (DTD) is a post-processing debug methodology that uses dedicated memory to store sample data for debug. It expands the storage capacity required for larger visibility windows by using external memory instead of local FPGA memory, like BRAM resources.

There are different DTD schemes that offer different speeds and visibility, according to the hardware resources they use. See the following for more information:

- [Comparison of Deep Trace Debug Schemes](#), on page 696
- [About Single-FPGA Deep Trace Debug \(DTD\)](#), on page 697
- [About Multi-FPGA DTD with a Debug Hub](#), on page 697
- [Debug Memory Summary for Deep Trace Debug](#), on page 699
- [Running Deep Trace Debug \(DTD\)](#), on page 702
- [Running Multi-FPGA Built-in DTD \(HAPS-80\)](#), on page 711
- [DTD Cross-Trigger](#), on page 752
- [Using a Debug Hub with SATA \(DTD2\)](#), on page 716
- [Setting up Hardware for DTD2 with HAPS-70](#), on page 731
- [HAPS-80 DTD Link Training Failures and Recommended Actions](#), on page 701

HAPS-100 systems include built-in DDR4 memory on daughter cards that can be used for storing sample data. By default, each HAPS-100 FPGA has a daughter card installed.

The following topics in the *HAPS Prototyping Instrumentor User Guide* describe the DTD schemes available for HAPS-100 systems:

- [Setting up DTD for HAPS-100 Designs](#), on page 108
- [Specifying the DTD Buffer for HAPS-100 Designs](#), on page 737
- [Setting up DTD Buffers for HAPS-100 Multi-FPGA Flows](#), on page 738
- [Limitations for HAPS-100 DTD](#), on page 115

Comparison of Deep Trace Debug Schemes

There are different DTD schemes, depending on the kind of storage memory used. Storage memory could be external or internal or built-in. This table summarizes the different DTD capabilities available.

| System | Capability | Description |
|---|--|--|
| DTD (Single-FPGA Deep Trace Debug) | | |
| HAPS-70 | <ul style="list-style-type: none"> Max Speed: 250 signals @140 MHz | Single-FPGA instrumentation |
| HAPS-80 | <ul style="list-style-type: none"> Max Visibility: 2042 signals @ 17.5 MHz | Running Deep Trace Debug (DTD), on page 702 |
| DTD2 (Multi-FPGA DTD with External Debug Hub and SATA) | | |
| HAPS-70 | Single system, multi-FPGA: <ul style="list-style-type: none"> Max Speed: 32 signals/link @ 70 MHz Max Visibility: 255 signals/link @ 8.75 MHz | Multi-FPGA instrumentation Total of 8 SATA links to support up to 8 FPGAs Using a Debug Hub with SATA (DTD2), on page 716 |
| DTD3 (Multi-FPGA DTD with External Debug Hub and QSFP) | | |
| HAPS-70 | Single system, multi-FPGA: | Multi-FPGA instrumentation |
| HAPS-80 | <ul style="list-style-type: none"> Max Speed: 128 signals/link @ 140 MHz Max Visibility: 1000 signals/link @ 17.5 MHz Per hub: <ul style="list-style-type: none"> 4000 signals/FPGA x 1 FPGA 1000 signals/FPGA x 4 FPGAs | Total of 4 QSFP links per hub to support up to 4 FPGAs Support for 2 chained hubs Running DTD2 with a HAPS-DX7 System and SATA Cabling, on page 718 |
| DTD 4 (Built-In Dedicated DDR3 Memory) | | |
| HAPS-80 | Single system, multi-FPGA: <ul style="list-style-type: none"> Max Speed: 64 signals/FPGA @140 MHz Max Visibility: 512 signals @ 17.5 MHz | <ul style="list-style-type: none"> Single FPGA instrumentation Multi-FPGA instrumentation (up to 4) Running Multi-FPGA Built-in DTD (HAPS-80), on page 711 |

About Single-FPGA Deep Trace Debug (DTD)

Single-FPGA HAPS Deep Trace Debug (DTD) lets you run RTL debug or post-compile debug on a single FPGA. In the multi-FPGA context, single-FPGA debug refers to debug of a partitioned individual FPGA. You might want to debug a single FPGA for to save runtime. There are some differences in the memory storage used with different HAPS systems. With HAPS-70 systems, you use an external dual-rank or single-rank HT3 DDR3 daughter board. With HAPS-80 systems, you use the built-in DDR3 memory for debug storage.

The following table summarizes the key features:

| Debug Interface | UMRBus |
|--------------------------------|---|
| Sample Buffer Memory | Dual Rank HT3 DDR3 daughter card Single Rank HT3 DDR3 daughter card |
| Number of Instrumented Signals | Up to total 2042 signals instrumented |
| Maximum Sample Frequency | Up to 250 signals: 140 MHz Up to 506 signals: 70 MHz Up to 1018 signals: 35 MHz Up to 2042 signals: 17.5 MHz |
| Over-Instrumentation | MUX set of 8 signal groups for up to 16K signals. Each signal group can have up to 2042 signals. |

About Multi-FPGA DTD with a Debug Hub

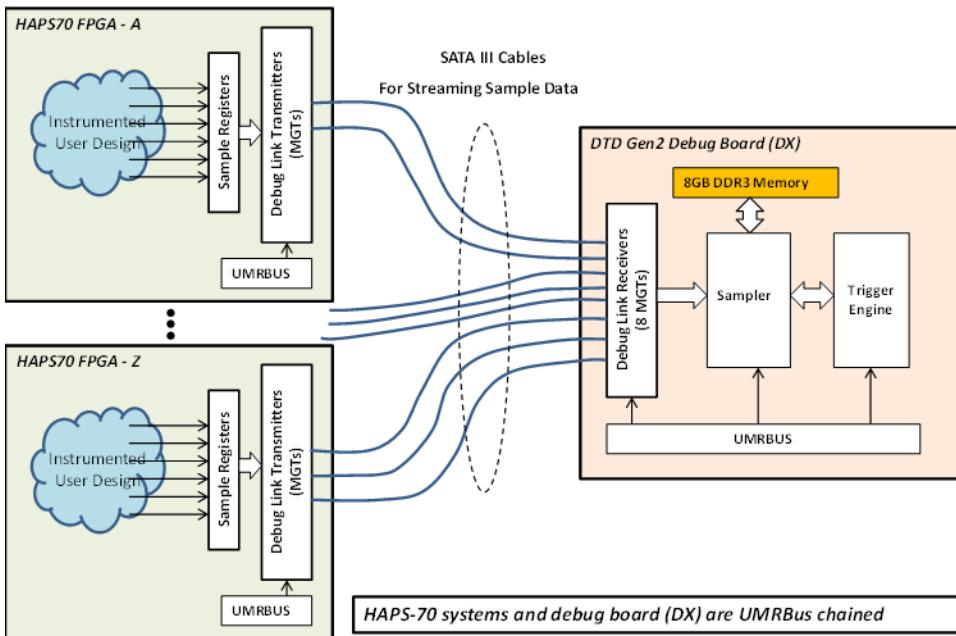
One way to run RTL debug on a prototype that is realized on multiple FPGAs is to use external memory as debug storage. This requires additional hardware. The following external memory storage scenarios are supported:

- [HAPS-DX7 Debug Hub with SATA Connections](#), on page 697
- [HAPS-DX7 Debug Hub with QSFP](#), on page 699

HAPS-DX7 Debug Hub with SATA Connections

This scheme uses a single HAPS-DX7 system as a debug hub to capture sample data, to trigger it. It provides up to 8 GB of memory. You can have up to eight FPGAs per debug hub. DTD2 uses only the MGB connectors on the HAPS-70 systems, leaving all the HT3 connectors free for design applications. This debug scheme is based on a SATA cable connection.

For details, refer to [Running DTD2 with a HAPS-DX7 System and SATA Cabling, on page 718](#).



The following table summarizes the key features:

| | |
|---------------------------------------|---|
| Debug Interface | UMRBus |
| Prototyping Hardware Platform | HAPS-70 |
| Debug Hub Hardware Platform | HAPS-DX7 |
| Debug Link Media | Synopsys edition SATA-II cross-over cable |
| Number of FPGAs | Up to 8 per debug hub (8 per debug session) |
| Number of Instrumented Signals | Up to 2042 instrumented signals per debug hub, including sample and trigger signals |

| | |
|---|--|
| HAPS-70 Maximum Sample Frequency (SATA cable link) | Up to 32 bits per MGT debug link: 70 MHz Up to 64 bits per MGT debug link: 35 MHz Up to 128 bits per MGT debug link: 17.5 MHz Up to 256 bits per MGT debug link: 8.75 MHz |
| HAPS-DX7 Maximum Sample Frequency | Up to 250 signals: 140 MHz Up to 506 signals: 70 MHz Up to 1018 signals: 35 MHz Up to 2042 signals: 17.5 MHz |
| IICE Support | 1 per debug implementation |

Limitations:

- Compiled netlist input is used at the individual FPGA level, which means that simulation with the instrumented design is not supported
- You can only have a single IICE
- Mux groups are not supported
- No trigger import and export

HAPS-DX7 Debug Hub with QSFP

This scheme uses a single HAPS-DX7 system as a debug hub but uses QSFP connections instead of SATA cables. See [Using a Debug Hub with QSFP Connectors, on page 740](#) for more information.

Debug Memory Summary for Deep Trace Debug

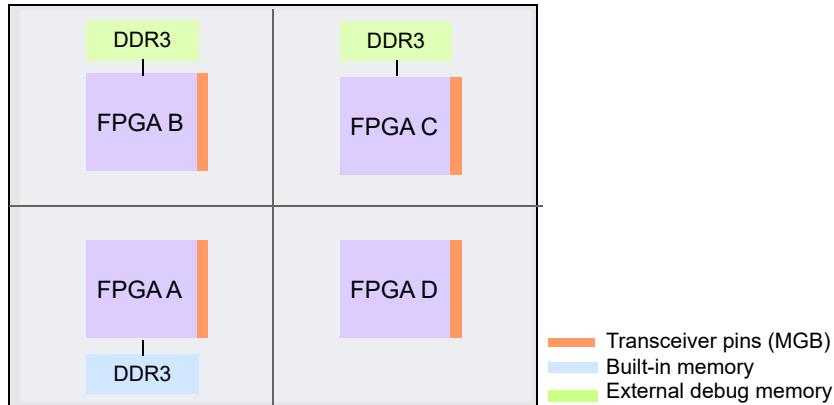
The following table summarizes the memory storage options for deep trace debug.

Except for the built-in option, all others require additional hardware.

| System & Buffer Storage | Description |
|--|---|
| HAPS-70/80 with HAPS DDR3_SODIMM2R_HT3 or HAPS DDR3_SODIMM_HT3 (DTD) | 8 GB per HAPS-70 daughter board. See Running Deep Trace Debug (DTD) , on page 702. |
| HAPS-70/80 with single HAPS-DX7 Debug Hub (DTD2) | See Running DTD2 with a HAPS-DX7 System and SATA Cabling , on page 718 |
| HAPS-70/80 with QSFP and 1-4 HAPS-DX7 systems (DTD3) | See Using a Debug Hub with QSFP Connectors , on page 740. |
| HAPS-80 with built-in storage (DTD4) | See Running Multi-FPGA Built-in DTD (HAPS-80) , on page 711. |
| HAPS-80 with external storage (DTD5) | See DTD Cross-Trigger , on page 752. |

External DDR3 Memory

You can add DDR3 memory for debug storage as needed. The following figure shows additional DDR3 memory added to debug FPGAs B and C:



HAPS-80 DTD Link Training Failures and Recommended Actions

The DTD link training may fail due to various reasons. A prerequisite to using the DTD link training feature is to use a direct cabling between the hub FPGAs for multi-hub configuration.

DTD link training fails during part 1

- Check if the user-specified sampling frequency is greater than the frequency provided in the instrumentation.log through Confpro for the particular design.
- Check if for timing violations in the design and IP.
- Check for improper cable connections while using DTD2 and DTD3 buffer types; for example, faulty HT3 cable.

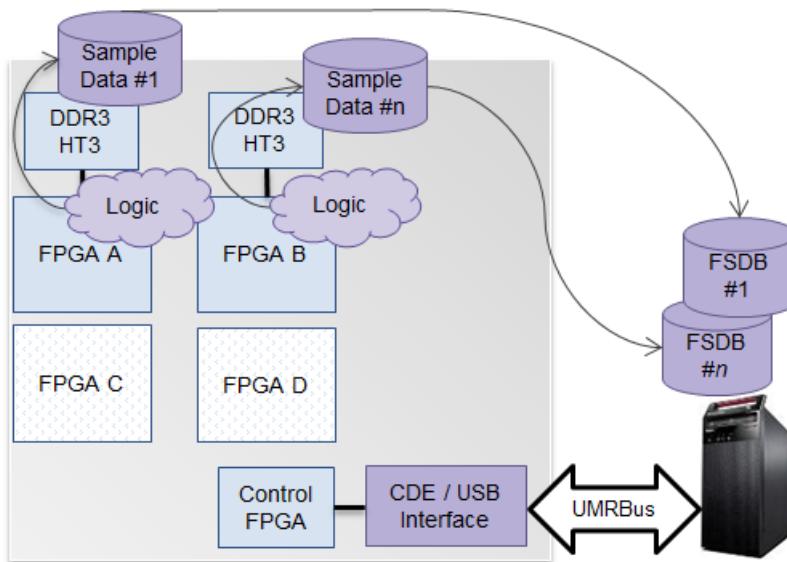
The DTD link training fails during part 2

- Check for timing violations in transceiver user clocks in the HUB FPGA. Check clocks iice_sclk_srcbufr, iice_sclk_local, gthe3_rx_2 for timing violations in the timing report.
- Check for timing violations in user or DTD internal clock in instrumented FPGAs. For DTD modules, check clocks dtd2dtxclk, gthe3_tx_1 for timing violations.
- Check for skewed user-clocks between systems that can cause misalignment in the receiver channel bonding check.
- Check for timing violations in max delay constraints for DTD2 reset.
- Check for multi-hub designs with no sample clock available through Confpro for all instrumented boards.

Running Deep Trace Debug (DTD)

Single-FPGA HAPS Deep Trace Debug (DTD) lets you run RTL or post-compile debug on a single FPGA. You can also use it to debug a single partitioned FPGA in a multi-FPGA design. This DTD scheme uses external DDR3 or DDR4 (HAPS-80 only) memory for debug storage.

The following figure shows the tool process to run debug on single FPGAs using a HAPS-70 or HAPS-80 system with DDR3 memory. The HT3 DDR3 provides sample storage, up to 8 GB for each FPGA. Note that each FPGA is a separate debug process and that separate FSDB databases are generated for individual FPGA debug runs.



See the following topics for details:

- [Running Single-FPGA Debug on a HAPS-80 FPGA](#), on page 703
- [Running Single-FPGA Debug on HAPS-70 with DDR3](#), on page 707
- [Running the RTL Debugger](#), on page 681

Features for DTD with a DDR3 Daughter Board

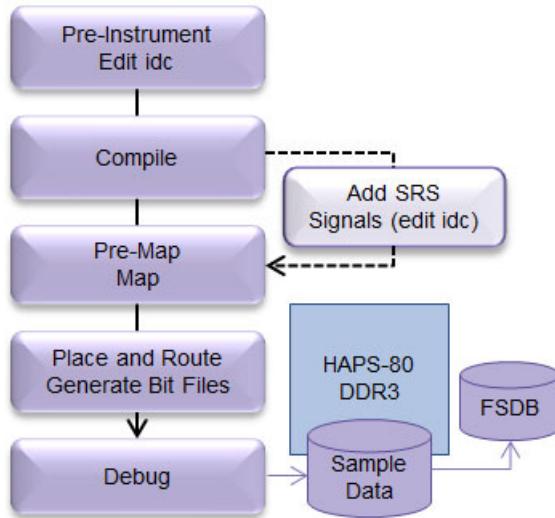
The following table summarizes the key features. The numbers apply to DTD with a DDR3 daughter board.

| Debug Interface | UMRBus |
|--------------------------------|---|
| HAPS-80 Sample Buffer Memory | Built-in DDR3 memory on the HAPS-80 system Dual Rank HT3 DDR3 daughter card Single Rank HT3 DDR3 daughter card |
| HAPS-70 Sample Buffer Memory | Dual Rank HT3 DDR3 daughter card Single Rank HT3 DDR3 daughter card |
| Number of Instrumented Signals | Up to total 2042 signals instrumented |
| Maximum Sample Frequency | Up to 250 signals: 140 MHz Up to 506 signals: 70 MHz Up to 1018 signals: 35 MHz Up to 2042 signals: 17.5 MHz |
| Over-Instrumentation | MUX set of 8 signal groups for up to 16K signals. Each signal group can have up to 2042 signals. |

Running Single-FPGA Debug on a HAPS-80 FPGA

Single-FPGA DTD mode uses a memory-based DDR3 daughter board to extend the depth of the sample memory and save FPGA memory resources and GPIOs for design use. Use the DDR3 memory module supplied on FPGA A of HAPS-80 systems for single-FPGA debug. To run single-FPGA debug on one of the other FPGAs, add a DDR3 or DDR4 memory module.

The following figure summarizes the flow to run debug on a single HAPS-80 FPGA, after it has been partitioned. You can run debug on the RTL before compile (pre-instrument stage) or on the compiled database. The dotted line indicates post-compile instrumentation. For more about each step shown, refer to [Launching and Running the Debugger, on page 681](#).

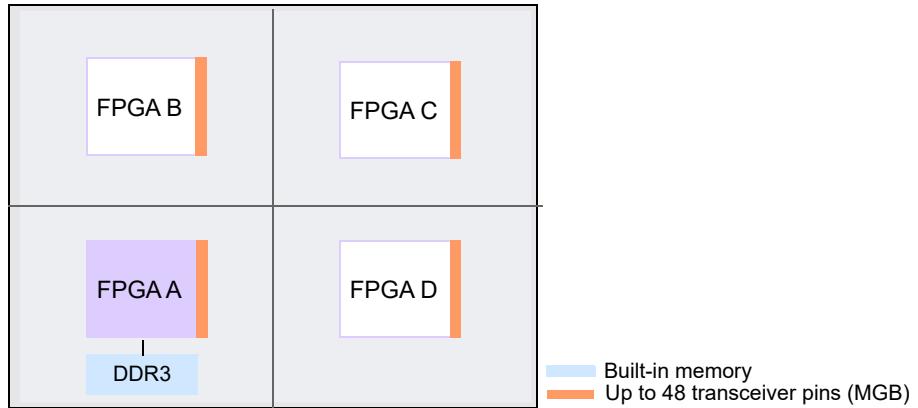


1. In the prototyping software, load a database and use the run pre_instrument command to create a database state for instrumentation:

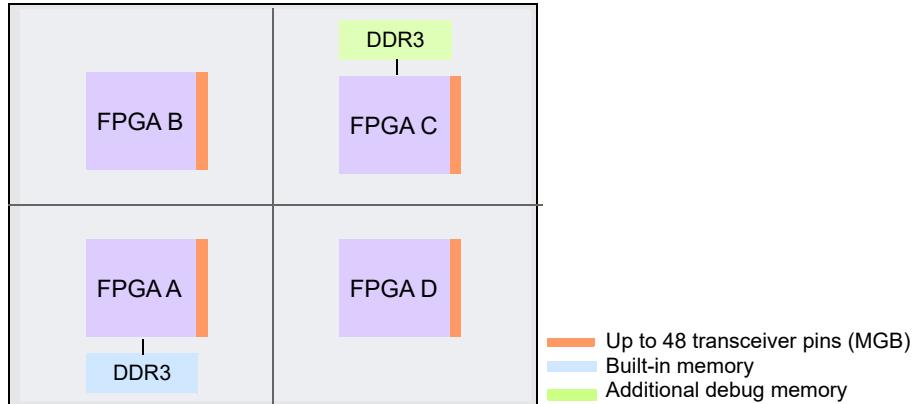
root → *run pre_instrument* → r0

2. Do the following to run debug on FPGA A:

- Connect to J1, J2, and J3 on FPGA A, without changing this setup; only these locations are supported for DTD. The HAPS-80 system includes a DDR3 memory module on connector locations A1-A3. It is recommended that you do not remove or move the debug DDR3_SODIMM2R_HT3 from its default location.



3. To run debug on FPGAs B, C, or D, add external memory, as described below.
 - Add an external DDR3 or DDR4 daughter board for debug sample storage. The following figure shows DDR3 memory added to FPGA C.
 - Connect the memory module to locations A1-A3 on the FPGA.



4. Instantiate the built-in System IP.
5. Run edit idc and configure the memory with the iice sampler Tcl commands, or click the IICE icon () in the instrumentor GUI and set these options on the IICE Sampler tab.

When you run debug on a partitioned FPGA, you create another idc file at the individual FPGA level which applies to the individual FPGA only, in addition to the top-level idc file.

- Configure the buffer type with this iice sampler command:
- iice sampler -iice haps_DTDFor a DDR4 daughter card, specify the memory type with -ram. DDR3 memory is the default, so you do not have to specify the memory type if you are using it. For example:

```
iice sampler -iice {i2042} -ram {type DDR4_8G}
```

- Configure the depth of the buffer. The value you set cannot be increased later when you run debug.

```
iice sampler -depth value
```

6. Define the HAPS hardware settings:

Alternatively, you can click the IICE icon in the instrumentor GUI and set these options on the DTD tab.

- Select the HAPS hardware system.
- Set the memory location.

7. Instrument the signals and sample clock.

Use the guidelines in the table below to determine the sample frequency based on the number of sample bits being instrumented. The maximum number of instrumented bits that can be sampled with DDR3 memory is 2042.

| Instrumented Bits | Max Sample Frequency |
|-------------------|----------------------|
| 1 to 250 | 140 MHz |
| 251 to 506 | 70 MHz |
| 507 to 1018 | 35 MHz |
| 1019 to 2042 | 17.5 MHz |

- Save the file. The tool automatically calculates the maximum buffer depth based on the number of signals instrumented.

8. Specify the .idc file and run through the flow as usual:

- Add the signal list to the .idc file.

- Specify the .idc file with the run compile or run pre_partition commands.
 - Run through the rest of the partitioning and synthesis steps for the individual FPGAs as usual.
 - Export the files for debug as usual.
 - Make sure to copy the following Tcl scripts from the runtime directory when you export: mfhdtd_link.tcl (required for link training) and mfhdtd_values.tcl (setup file).
9. Initiate link training and run the debugger.
- Run the mfhdtd_link.tcl script, located in the exported runtime directory.
 - Check the log file for errors. The file reports the success of link training. In case of errors, follow the instructions and run the script in standalone confprosh shell for further debugging.
 - Once link training is complete, open the project in the debugger. Set trigger points and capture waveforms as usual.
 - Click Run. The tool generates the debug resets and synchronizes the system, independent of the user design resets. It is distributed to multiple FPGAs through the GCLK network or HT3 cables. By default, it uses fixed links between the FPGAs for debug. The on-board memory is used for buffering during debug. See [Running the RTL Debugger, on page 681](#) for details of the flow.
10. View the results.

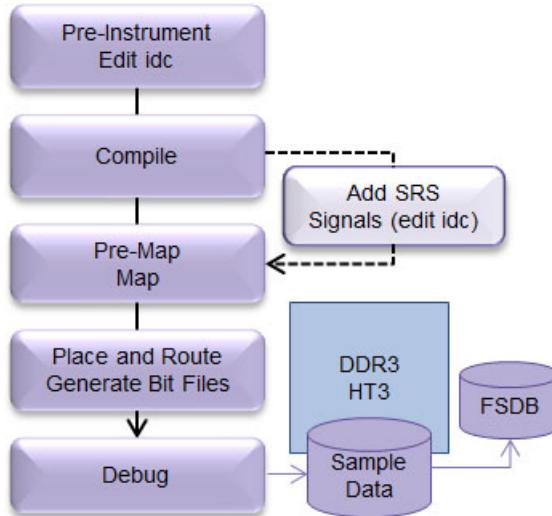
If the buffer depth is large, it can take a long time to download and view results. You can work around this, using these techniques:

- Vary the configured sample depth dynamically, from the minimum depth to maximum depth. See [Varying Buffer Depth Dynamically for DDR3, on page 687](#).
- View or write out a slice, instead of the entire waveform. See [Viewing Captured Deep Trace Debug Samples, on page 689](#).

Running Single-FPGA Debug on HAPS-70 with DDR3

This section describes the single-FPGA debug design flow with HAPS-70 systems. For specifics about using external DDR3 memory with HAPS-80 systems, see [Running Single-FPGA Debug on a HAPS-80 FPGA, on page 703](#).

The following figure summarizes the HAPS-70 flow to run single-FPGA debug on a partitioned FPGA. The dotted line indicates optional post-compile instrumentation. Use this flow to run debug individually for partitioned FPGAs.



Single-FPGA DTD mode uses a memory-based DDR3 daughter board to extend the depth of the sample memory and save FPGA memory resources and GPIOs for design use. The following steps describe how to configure and use a DDR3 daughter board for deep trace debug.

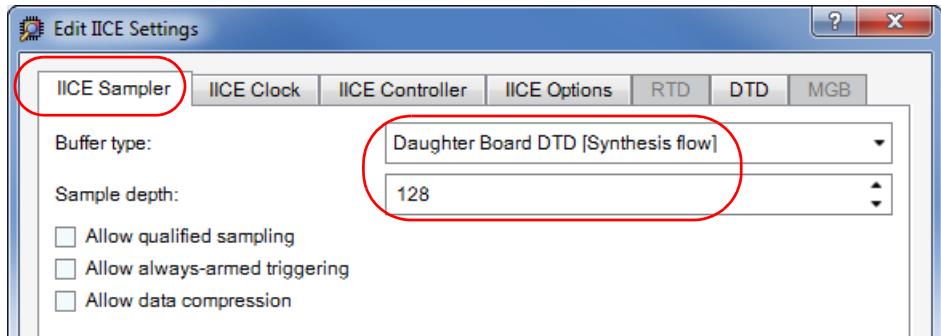
1. In the prototyping software, load a database and use the `run pre_instrument` command to create a database state for instrumentation:

`root → run pre_instrument → r0`

2. Run `edit idc` and configure the memory with the iice sampler Tcl commands:
 - Configure the buffer type. Use `haps_DTD` for HAPS-70 systems, and `DDR3 DTD` for HAPS-80 systems. This is the command syntax:
- `iice sampler {haps_DTD | DDR3 DTD}`
- Configure the depth of the buffer. The value you set cannot be increased later when you run `debug`.

`iice sampler -depth value`

Alternatively, click the IICE icon () in the instrumentor GUI and set these options on the IICE Sampler tab. Set Buffer type to Daughter Board DTD (Synthesis flow).



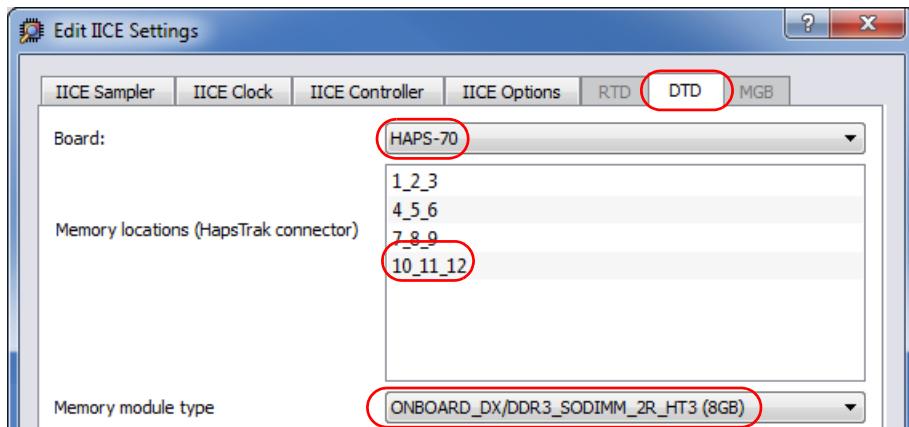
3. Define the HAPS hardware settings:

- Select the HAPS hardware board system:

`iice sampler -sram {board HAPS-70 | HAPS-80}`

- Set the memory location to the appropriate connector locations. It is recommended that you select adjacent locations that align with SLR layers. For HAPS-80, the only valid locations are 1, 2, and 3.

Alternatively, you can click the IICE icon in the instrumentor GUI and set these options on the DTD tab.



4. Instrument the signals and sample clock.

To maximize performance when using DDR3 memory, use the guidelines in the table below to determine the sample frequency based on the number of sample bits being instrumented. The maximum number of instrumented bits that can be sampled with DDR3 memory is 2042.

| Instrumented Bits | Max Sample Frequency |
|-------------------|----------------------|
| 1 to 250 | 140 MHz |
| 251 to 506 | 70 MHz |
| 507 to 1018 | 35 MHz |
| 1019 to 2042 | 17.5 MHz |

- Save the file. The tool automatically calculates the maximum buffer depth based on the number of signals instrumented, as shown in the preceding table.

5. Set up and connect the HAPS system and daughter board for debug.

See [Hardware Components for DTD2 \(with HAPS-70\), on page 732](#) and the appropriate hardware documentation for details.

6. Run RTL debug ([Running the RTL Debugger, on page 681](#)).

The software uses the on-board memory for increased buffering capacity during debug.

7. View the results.

If the buffer depth is large, it can take a long time to download and view results. You can work around this, using these techniques:

- Vary the configured sample depth dynamically, from the minimum depth to maximum depth. See [Varying Buffer Depth Dynamically for DDR3, on page 687](#).
- View or write out a slice, instead of the entire waveform. See [Viewing Captured Deep Trace Debug Samples, on page 689](#).

Running Multi-FPGA Built-in DTD (HAPS-80)

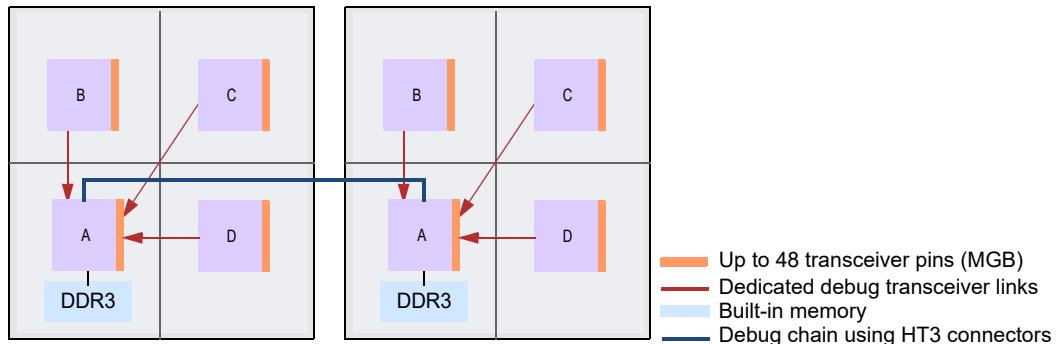
You can use the built-in DDR3 memory included on FPGA A of HAPS-80 systems to run debug. Using built-in memory offers at-speed signal visibility without extra hardware. For example, you can have a large sample window with 64 signals per FPGA at a maximum speed of 160 MHz. Or you can increase visibility to 1024 signals per FPGA, running at 10 MHz. HAPS-80 debug with built-in memory storage is also referred to as DTD4.

See the following for details:

- [Use Model for Multi-System Multi-FPGA Debug with Built-in DTD](#), on page 711
- [Running Multi-FPGA Debug with HAPS-80 Built-in Memory](#), on page 712

Use Model for Multi-System Multi-FPGA Debug with Built-in DTD

In this chained system scenario, the built-in DDR3 memory on each FPGA A is used for debug storage.



The following table shows the capacity per HAPS-80 system, if the tagged debug signals are evenly distributed between FPGAs.

| Number of Signals | Maximum Sample Clock Frequency (MHz) |
|-------------------|--------------------------------------|
| 250 | 140 |

| Number of Signals | Maximum Sample Clock Frequency (MHz) |
|-------------------|--------------------------------------|
| 506 | 70 |
| 1018 | 35 |
| 2042 | 17.5 |

Each FPGA is connected to FPGA A through MGB links. The bandwidth of the MGB links is 128 bits at 140 MHz. If tagged signals are not spread out but concentrated on one FPGA, the tool uses the maximum mux ratio instead of the values listed in the table. Calculate the mux ratio for a single FPGA using this formula:

$$\text{Single FPGA mux ratio} = \text{tagged signal bits} / 128$$

For example, say FPGA C has the most tagged signals. If there are 435 bit signals, the mux ratio for FPGA C is $435/128 = 4$. The maximum sample frequency is then $140/4 = 35$ MHz, not 70 MHz as in the table. For multiple HAPS-80 systems with tagged signals, the tool uses the highest mux ratio between hubs.

Further, if all the tagged debug signals are located on FPGA A, the tool converts the DTD builtin IICE to single-FPGA DTD IICE. This means that the MGB links are not used, so the maximum sample frequency is higher.

Running Multi-FPGA Debug with HAPS-80 Built-in Memory

For HAPS-80 systems you can use built-in memory as a debug resource, instead of external memory like a debug hub. To accomplish this, the system architecture includes built-in memory on FPGA A. This DTD scheme is also called DTD4.

The following procedure describes how to use the built-in memory to debug the multi-FPGA usage scenarios described in [Use Model for Multi-System Multi-FPGA Debug with Built-in DTD, on page 711](#). You can use the memory on FPGA A for the debug all the FPGAs on a single system, or use FPGA A memory resources from multiple systems as debug memory when systems are chained together.

1. Set up your design so that FPGA A can be used for debug.

- If you do not set up any part of the user design on FPGA A, it can be dedicated entirely to debug with the receive IP on FPGA freely sending and receiving debug signals from the other FPGAs.
 - If FPGA A contains part of the user design but does not include instrumented signals, the tool uses the receive IP on FPGA A to communicate with the other FPGAs for debugging.
 - If FPGA A contains part of the user design and includes instrumented signals, the tool inserts both send and receive IP.
 - The built-in debug memory is a DDR3 daughter card that is connected to FPGA A. FPGA A includes a centralized IICE and the DDR3 interface. It communicates with the other FPGAs on the HAPS system using the eight built-in GTX links and the transceiver pins on each FPGA (up to 48). Use the default location and connect to J1, J2, and J3 on FPGA A, without changing this setup.
2. Manually define the IICE settings in the .idc file.
- Use the appropriate IICE sampler setting, according to whether the built-in memory being used is local or not:

| Debug | iice sampler -iice Setting |
|--|-----------------------------------|
| Single-FPGA, for FPGA A (local memory) | haps_dtd |
| Single-FPGA, for FPGAs B, C, or D | haps_dtd |
| Multi-FPGA | haps_dtd_builtin |

Do not use the export trigger command (`iice controller -iice {IICE_NAME} -exporttrigger 1`) with `haps_dtd_builtin`, because this command assigns the pin to BD14, and there might be a resource conflict if your design uses the same pin. There is a large cross-trigger latency for all transceiver-based IICE types. Instead, use multi-FPGA BRAM to export triggers.

- For multi-FPGA logic, include control logic parameterization. The following example shows assignments for multi-FPGA debug on a single system:

```

device jtagport umrbus
iice new {IICE_user} -type regular
iice controller -iice {IICE_user} none
iice sampler -iice {IICE_user} haps_dtd_builtin
iice sampler -iice {IICE_user} -dtd_clock {GCLK3}
iice sampler -iice {IICE_user} -dtd_reset_type {AUTO}
iice mgb -iice {IICE_user} -add_hub {fb1.uA}
iice sampler -iice {IICE_user} -depth 33554431
iice sampler -iice {IICE_user} -pipe 3

```

The DTD clock frequency must be set to 175MHz. The DTD clock is specified with command:

```
iice sampler -iice {IICE_user} -dtd_clock {GCLK}
```

| Specify | Option |
|----------------------------|------------------|
| Buffer type | haps_dtd_builtin |
| DTD reference clock | -dtd_clock |
| Type of reset distribution | -dtd_reset_type |
| Debug hub | -add_hub |

- Set the reset distribution scheme with iice sampler -dtd_reset_type. Set it to AUTO to use HT3 cables, or GCLK to use the global clock resources. In both cases, add the dtd_reset option to the iice sampler command to specify the source FPGA or GCLK resource to use.

If your design does not use all the available HAPS FPGAs (for example, if the design only uses FPGAs A and B but not C or D on a HAPS-80 S104 system), do not use dtd_reset_type AUTO for reset replication. This could cause an error during system route because there are no HT3 cables connecting FPGAs C and D. Instead, do the following:

First lock the unused FPGAs that are not part of the design (PCF):

```

bin_attribute fb1.uc -locked
bin_attribute fb1.uD -locked

```

Then restrict reset distribution to the design FPGAs only, by setting -dtd_reset_bins in the idc file to just those FPGAs:

```
iice sampler -iice {IICE_DTD4} -dtd_reset_bins {fb1.uA fb1.uB}
```

- Instantiate one hub per system.
3. Complete the .idc file and specify it.
 - Add the signal list to the .idc file.
 - Specify the .idc file with the run compile or run pre_partition commands.
 4. Complete partitioning, synthesize individual FPGAs, and export files.
 - Run through the rest of the partitioning and synthesis steps for the individual FPGAs as usual.
 - Export the files for debug as usual.
 - Make sure to copy the following Tcl scripts from the runtime directory when you export, as they are needed for link training: mfhdtd_link.tcl (required for link training) and mfhdtd_values.tcl (setup file).
 5. Initiate link training by running the link training script.
 - Run the mfhdtd_link.tcl script, located in the exported runtime directory. For example:

```
confprosh mfhdtd_link.tcl
```
 - Check the log file for errors. The file reports the success of link training. In case of errors, follow the instructions and run the script in standalone confprosh shell for further debugging.

Note that you must do link training each time you modify the configuration settings. Proceeding without link training might result in misaligned waveforms.
 6. Run the debugger.
 - Once link training is complete, open the project in the debugger. Set trigger points and capture waveforms as usual.
 - Click Run. The tool generates the debug resets and synchronizes the system, independent of the user design resets. It is distributed to multiple FPGAs through the GCLK network or HT3 cables. By default, it uses fixed links between the FPGAs for debug.

Using a Debug Hub with SATA (DTD2)

The DTD2 multi-FPGA debug scheme uses a HAPS-DX7 system for debug storage memory. The following links provide more information:

- [Running DTD2 with a HAPS-DX7 System and SATA Cabling](#), on page 718
- [Running the RTL Debugger](#), on page 681
- [Setting up Hardware for DTD2 with HAPS-70](#), on page 731
- [Configuring the Sample Clock for DTD2](#), on page 723
- [Working with Maximum Sample Frequency for DTD2](#), on page 724
- [Configuring a Reference Clock for DTD2](#), on page 725
- [Configuring Clocks on the HAPS-DX7 System for DTD2](#), on page 727
- [Using Xilinx Backbone Clock Routes with DTD2](#), on page 727
- [Configuring Resets for DTD2](#), on page 728
- [Allocating Locations for PLLs and MMCMs](#), on page 730

Comparison of DTD and DTD2 for HAPS-70 Systems

You can run single-FPGA or multi-FPGA debug with the HAPS-70 hardware, using external memory for storage. The table summarizes some of the hardware differences between the two modes:

| | Single FPGA (DTD) | Multi-FPGA (DTD2) |
|-------------|---|--|
| FPGAs | 1 | Up to 8 |
| Capacity | Up to 8 GB | 8 GB |
| Sample rate | 250 bits @ 100 MHz 500 bits @ 50 MHz | 32 bits per link @ 60 MHz 64 bits per link @ 30 MHz |
| Memory | 4 GB DDR3 HT3 DB 8 GGB DDR3 DB | SODIMM on HAPS-DX7 |

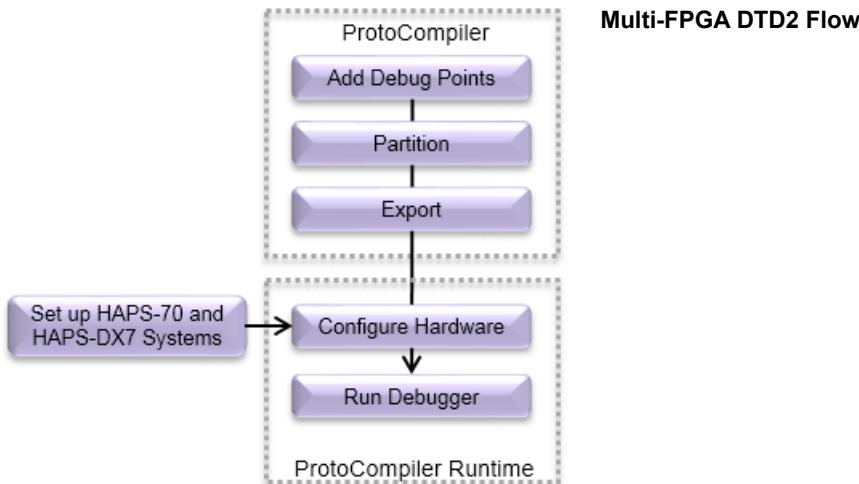
Multi-FPGA Debug Hub Flow for HAPS-70 Systems

The figure summarizes how to run multi-FPGA deep trace debug with a debug hub for DTD2. For details about the steps in the flow, see [Running DTD2 with a HAPS-DX7 System and SATA Cabling, on page 718](#).

For flow details, follow these links:

- [Multi-FPGA Debug Hub Flow for HAPS-70 Systems, on page 717](#)
- [Running DTD2 with a HAPS-DX7 System and SATA Cabling, on page 718](#)
- [Running the RTL Debugger, on page 681](#)
- [Setting up Hardware for DTD2 with HAPS-70, on page 731](#)
- [Configuring the Sample Clock for DTD2, on page 723](#)
- [Working with Maximum Sample Frequency for DTD2, on page 724](#)
- [Configuring a Reference Clock for DTD2, on page 725](#)
- [Configuring Clocks on the HAPS-DX7 System for DTD2, on page 727](#)
- [Using Xilinx Backbone Clock Routes with DTD2, on page 727](#)
- [Configuring Resets for DTD2, on page 728](#)
- [Allocating Locations for PLLs and MMCMs, on page 730](#)

Running DTD2 with a HAPS-DX7 System and SATA Cabling



The following procedure describes one way to run deep trace debug with HAPS-70 systems, using Multi-FPGA DTD2. DTD2 uses a HAPS-DX7 system for memory storage during the debug process. The HAPS-DX system provides 8 GB for debug storage. It provides HAPS real-time debug and uses Siloti® data interpolation and data expansion. Information is transmitted through SATA crossover cables and MGB (multi-gigabit) connectors, so no HapsTrak 3 connectors are needed. The DTD2 setup uses UMRBus for debug board control and access, and a clock coax cable for synchronization. The sampling frequency is 500 bits at 30 MHz or 250 bits at 60 MHz.

1. Set up the hardware.

The multi-FPGA DTD hardware setup requires riser MGB cards, SATA 4 HS MGB cards, SATA crossover cables, a clock coax cable, and a HAPS-DX7 system as well as the HAPS-70/80 system.

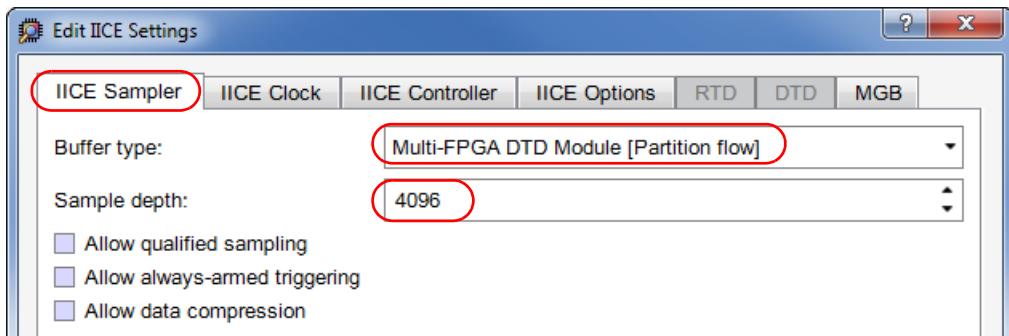
2. Get the design ready for instrumentation.

- Start the prototyping software and load a database.
- Specify the `run pre_instrument` command to create a database state for instrumentation:



- Run the `edit idc` command to start the instrumentor. If you specify the `-mode gui` argument to the command, you can work in the graphical interface.
 - Instrument the design as usual.
3. Configure the memory with the iice sampler Tcl commands:
- Configure the buffer type as `haps_DTD2`:
`iice sampler haps_DTD2`
 - Configure the depth of the buffer: `iice sampler -depth value`

Alternatively, click the IICE icon () in the instrumentor GUI and set options on the IICE Sampler tab. Set Buffer type to Multi-FPGA DTD Module, and set the sample depth.

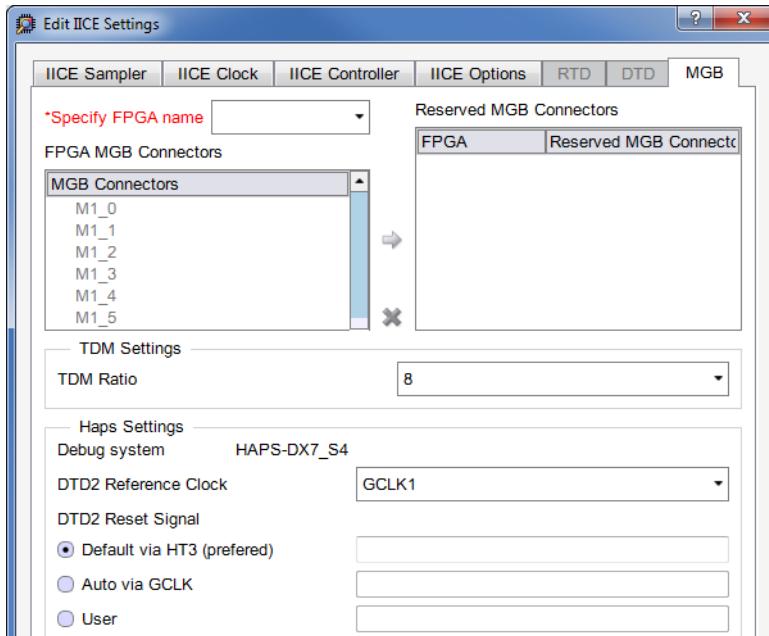


The instrumentation is sent to multi-gigabit transmitters on the FPGA instead of IICE, and sent to the HAPS-DX system over the cables. Note that the cables are only for electrical transmissions. The SATA protocol is not used; a proprietary protocol is used instead.

4. Configure the hardware connections.
- On the MGB tab, specify the FPGA to configure in **FPGA name**.
 - Specify the connections reserved for design use. To do this, click **Show connectors** to display the available connections for the FPGA in the panel on the left. Select the ones to reserve and click the right arrow to move them to the right panel. The tool uses non-reserved connections for debugging. To remove a connector from the reserved list, select it in the right panel and click the X icon (Remove selected reserved MGB connectors).

Alternatively you can use a command instead of the GUI, as shown in the following example. The empty set of curly brackets {} indicates that no connections were reserved for that FPGA:

```
iice mgb -mgb_reserved {{ } {MGB1.J2.X2 MGB1.J2.X3 MGB1.J2.X4 }}
```



- Set the TDM ratio. Enable Auto TDM factor to automatically set the TDM ratio. Alternatively, disable Auto TDM factor, and specify a value for TDM factor. The tool updates the Max user frequency field, based on your design and the ratio you specified. A higher mux factor means a lower sampling frequency.

This is an example of the commands you can use instead of the GUI:

```
iice mgb -mgb_muxfactor 2
iice sampler -DTD2_reset {resetA} -DTD2_clock {clkA}
```

- Set the reference clock to GCLK[1-7, 9-12]. The frequency of the debug reference clock locks the transmit/receive transceivers.
- To maximize deep sample buffer benefits, use FSDB, not vcd. The latter is ASCII-text based, and not suitable. If you need to work with smaller files, create a smaller FSDB by slicing the design and isolating the parts you want to debug.

- Save the idc file.
5. Complete the rest of the prototyping flow and generate FPGAs.
- Use the run compile, run pre_partition and run partition commands to compile and partition the design.
 - Generate FPGAs with the run system_generate command. The tool reports the cabling required for DTD, and generates scripts to run it.
 - Use the view report command to open the log file created after the FPGAs are generated, and check the cabling setup information.

This file reports information as shown in this example:

```
DTD2 summary:  
QN: Z100 |*****  
QN: Z100 | Instrumentation Memory Connections: HAPS-DX7_S4  
QN: Z100 | 8 Connections Total:  
QN: Z100 |   MGB1.J2.X5 - Must be connected  
QN: Z100 |   3 of the following 7 must be connected  
QN: Z100 |     MGB1.J2.X2  
QN: Z100 |     MGB1.J2.X3  
QN: Z100 |     MGB1.J2.X4  
QN: Z100 |     MGB1.J3.X5  
QN: Z100 |     MGB1.J3.X2  
QN: Z100 |     MGB1.J3.X3  
QN: Z100 |     MGB1.J3.X4  
QN: Z100 | User Design Connections: HAPS-70  
QN: Z100 | 8 Required Connections:  
QN: Z100 |   mb.uB:  
QN: Z100 |     MGB1.J2.X5  
QN: Z100 |     CB1.J2.X2  
QN: Z100 |     MGB1.J2.X3  
QN: Z100 |     MGB1.J2.X4  
QN: Z100 |   mb.uA:  
QN: Z100 |     MGB1.J2.X5  
QN: Z100 |     MGB1.J2.X2  
QN: Z100 |     MGR1.J2.X3  
QN: Z100 |     MGB1.J2.X4  
QN: Z100 | MGB connection mux ratio: 1  
QN: Z100 | Maximum sample clock frequency: 133.333333 MHz  
QN: Z100 |*****
```

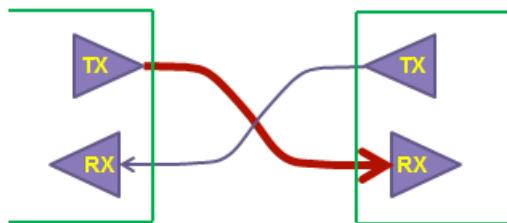
- Use the export runtime command to export the bit file.
6. Set up and connect the HAPS-70 and HAPS-DX7 systems for debugging.
- Do not use a Windows client server model.
- Start the ProtoCompiler runtime software.

- Open and edit the appropriate `run_dtd2_*.sh` script from the scripts in the `ProtoCompiler_runtime_install/lib/share/dtd2_scripts` directory. Select the script that matches the hardware debug system you are using and the number of FPGAs in your design. Edit it so that it matches your design and the connection information from the previous step. The script initializes all the FPGAs, loads the bit files, and sets up the hardware.
- For correct clock distribution, the HAPS global reset must be synchronized in a single place to GCLK0 and then synchronized to each user logic clock domain. Replicate and distribute the conditioned reset signal, not the synchronizer, for each clock domain to all the FPGAs in the system. Add pipeline registers to improve speed.

See [Using a Debug Hub with SATA \(DTD2\), on page 716](#) and the appropriate hardware documentation for details.

- Connect the cables as described in the system generate log. Note that Multi-FPGA DTD uses a proprietary protocol, not the SATA protocol. In the example shown in step 5, for instrumentation memory, you must connect MGB1.J2.X5 plus three other connections. The tool automatically links the rest. Make the connections listed under User Design Connections as described.

See [Using SATA Cable MGT Links, on page 734](#) for more information and [Hardware Connection Example for DTD2, on page 733](#) for an example. For further details, check the documentation for the hardware system you are using.



- Initialize the debug link (MGT) using confpro commands to configure the HAPS-DX reference clock to use PLLIN, and to initialize the debugger connection. See the Confpro documentation for details about the commands.
- Check the status LEDs on the HAPS-DX system. The system is ready to debug when LED1 and LED2 are green. If both LEDs are red,

check the setup and cabling, or check if debug link initialization failed.

7. Run RTL debug (*Running the RTL Debugger*, on page 681).

The software uses the connected hardware system as memory, for increased buffering capacity during debug.

8. View the results.

If the buffer depth is large, it can take a long time to download and view results. You can work around this, using these techniques:

- Vary the configured sample depth dynamically, from the minimum depth to maximum depth. See *Varying Buffer Depth Dynamically for DDR3*, on page 687.
- View or write out a slice, instead of the entire waveform. See *Viewing Captured Deep Trace Debug Samples*, on page 689.

Configuring the Sample Clock for DTD2

The sample clock for DTD2 must match the supported sample frequency. You can only have one sample clock, because you can only have one IICE per debug implementation.

- Make sure the sample clock has a frequency less than or equal to? the maximum sample frequency currently supported. The maximum sample frequency varies, depending on the number of instrumented signals. See the feature table in *About Multi-FPGA DTD with a Debug Hub*, on page 697 for details.
- Do not use GCLK0 as the sample clock for a DTD2 instrumented design, because this clock is at 100 MHz, which is higher than the maximum sample frequency currently supported (70 MHz when the debug link duplexing factor is 1).
- You can use a clock derived from GCLK0 as the sample clock, so long as its frequency is less than or equal to? the maximum sample frequency. For example, you can use a 50 MHz clock derived from GCLK0 in a 70 MHz design.

Working with Maximum Sample Frequency for DTD2

You can only have one sample clock, because of the current requirement of one IICE per debug implementation. In addition, the maximum sample frequency is determined by two factors:

- The MGT link with the maximum number of signals.
- The HAPS-DX7 debug hub system and its DDR3 memory, which determines the number of signals sampled and thus plays a part in determining the maximum sample clock frequency.

| | |
|---|--|
| HAPS-70 Maximum Sample Frequency (SATA cable link) | Up to 32 bits per MGT debug link: 70 MHz Up to 64 bits per MGT debug link: 35 MHz Up to 128 bits per MGT debug link: 17.5 MHz Up to 256 bits per MGT debug link: 8.75 MHz |
| HAPS-DX7 Maximum Sample Frequency | Up to 250 signals: 140 MHz Up to 506 signals: 70 MHz Up to 1018 signals: 35 MHz Up to 2042 signals: 17.5 MHz |

The maximum number of signals and sample frequencies is determined by a combination of the performance capability of the MGT links with the number of signals transmitted to the HAPS-DX7 debug hub and the memory storage requirements.

The following example shows how the number of MGT links can affect the maximum speed of a design partitioned into two FPGAs, with 256 instrumented signals in each partition:

- 4 SATA cables used per FPGA

In this case, the number of signals to be transmitted per SATA link is 64 bits per MGT. In this scenario, the SATA link can send signals sampled at up to 35 MHz. The HAPS-DX7 debug hub receives a total of 512 signals. If all the signals are to be stored in sample memory, the DDR3 sample storage capability limits the maximum sample frequency to 35 MHz (see the preceding table). Therefore, the maximum sample speed in this case is 35 MHz.

- 2 SATA cables used per FPGA

In this case, the number of signals to be transmitted per SATA link is 128 bits per MGT. In this scenario, the SATA link can send signals sampled at up to 17.5 MHz. The HAPS-DX7 debug hub receives a total of 512 signals. If all the signals are to be stored in sample memory, the

DDR3 sample storage capability limits the maximum sample frequency to 35 MHz (see the preceding table). Therefore, the maximum sample speed supported in this case is 17.5 MHz.

Configuring a Reference Clock for DTD2

Multi-FPGA HAPS DTD requires a 100 MHz common reference clock for HAPS-DX system debug hub and all the FPGAs with instrumented debug signals in the HAPS-70 systems. The HAPS-70 GCLK0 is not used for this purpose, so you must enable one of the configurable GCLKs to 100MHz.

The following examples show different scenarios where this required clock is defined.

Example 1: Two-FPGA Design

In this example there are two FPGAs, and GCLK3 of FB1 is configured to 100 MHz via PLL1_3, using the onboard 100 MHz oscillator as the reference clock.

1. Configure a GCLK to 100 MHz via PLL1_3, using the onboard 100 MHz oscillator as the reference clock.

To do so, specify the command shown below in the idc file during design instrumentation. The following example configures GCLK3, but you can use other GCLKs.

```
iice sampler -DTD2_clock {GCLK3}
```

2. Configure a HAPS-70 PLL out connector that connects to the PLL_IN connector of the HAPS-DX system to 100 MHz.

An SMB coax cable connects the PLL1_OUT HAPS-70 connector to the PLL_IN connector of debug hub HAPS-DX. For example, you can configure PLL1_4 of FB1, which corresponds to the PLL1_OUT SMB connector on the clock panel of HAPS-70 system, to 100 MHz. You could alternatively choose the PLL2_OUT SMB connector; if you do that, configure PLL2_4 of HAPS-70 to 100 MHz.

3. Configure GCLK on debug hub HAPS-DX (FB2):

- Configure GCLK1 (PLL1_1) to 100 MHz using PLLIN (100 MHz) as the reference clock.
- Configure GCLK3 (PLL1_3) to 100 MHz using PLLIN (100 MHz) as the reference clock.

Example 2: Three Chained HAPS-70 Systems

In this example FPGA prototyping setup, there are three chained HAPS-70 systems (FB1, FB2 and FB3), and a HAPS-DX debug hub at the end of the UMRBus chain (FB4). To construct a global clock structure in this setup, use FB1 as the master system for clock generation.

1. Configure a GCLK on FB1 to 100 MHz via PLL1_3, using the onboard 100 MHz oscillator as the reference clock.

To do so, specify the command shown below in the idc file during design instrumentation. The following example configures GCLK3, but you can use other GCLKs.

```
iice sampler -DTD2_clock {GCLK3}
```

2. Configure a HAPS-70 PLL out connector that connects to the PLL_IN connector of the HAPS-DX system to 100 MHz.

An SMB coax cable connects the PLL1_OUT HAPS-70 connector to the PLL_IN connector of debug hub HAPS-DX. For example, you can configure PLL1_4 of FB1, which corresponds to the PLL1_OUT SMB connector on the clock panel of HAPS-70 system, to 100 MHz. You could alternatively choose the PLL2_OUT SMB connector; if you do that, configure PLL2_4 of HAPS-70 to 100 MHz.

3. Configure a GCLK on FB2 to be sourced from the appropriate connector on FB1.

A CDE cable connects the CLK_RIGHT connector on FB1 to the CLK_LEFT connector on FB2. Configure GCLK3 of FB2 to be sourced from the CLK_LEFT connector, which is identical to GCLK3 of FB1.

4. Configure a GCLK on FB3 to be sourced from the appropriate connector on FB1.

A CDE cable connects the CLK_LEFT connector on FB1 to the CLK_RIGHT connector on FB3. Configure GCLK9 on FB3 to be sourced from the CLK_RIGHT connector, which is identical to GCLK3 on FB1 and FB2. Currently, the run system_generate command does not automatically take this into account, so you must add a segment to your ProtoCompiler TCL script. See [Example: Tcl Script Segment to Account for Different GCLKs, on page 738](#) for an example.

5. Configure GCLK on debug hub HAPS-DX (FB4):

- Configure GCLK1 (PLL1_1) to 100 MHz using PLLIN (100 MHz) as the reference clock.
- Configure GCLK3 (PLL1_3) to 100 MHz using PLLIN (100 MHz) as the reference clock.

Configuring Clocks on the HAPS-DX7 System for DTD2

You do not need to configure the user design clock or sample clock on the HAPS-DX7 debug hub. The clock configuration on the debug hub is independent of the user design clock. You only need to configure the reference clocks as follows:

- Configure GCLK1 (PLL1_1) to 100 MHz using PLLIN (100 MHz) as the reference clock.
- Configure GCLK3 (PLL1_3) to 100 MHz using PLLIN (100 MHz) as the reference clock.

Using Xilinx Backbone Clock Routes with DTD2

Some clock rules may cause sub-optimal placement messages in Vivado place and route and a recommendation to override the setting. For example:

ERROR: [Place 30-575] Sub-optimal placement for a clock-capable IO pin and MMCM pair. If this sub optimal condition is acceptable for this design, you may use the CLOCK_DEDICATED_ROUTE constraint in the .xdc file to demote this message to a WARNING. However, the use of this override is highly discouraged. These examples can be used directly in the .xdc file to override this clock rule.

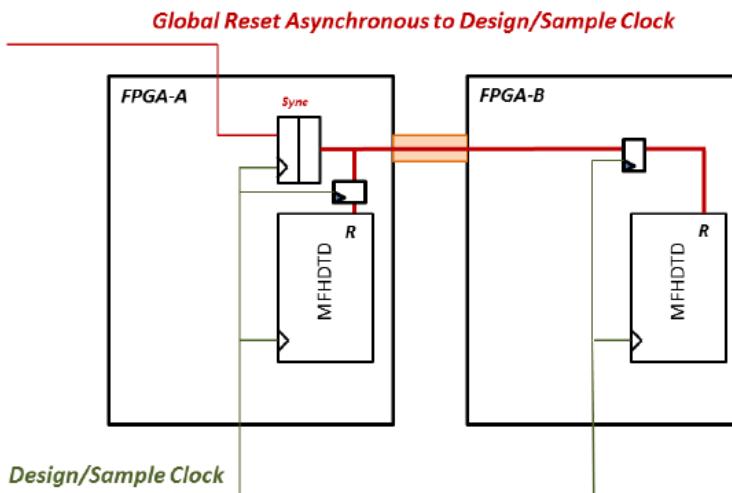
```
<set_property CLOCK_DEDICATED_ROUTE BACKBONE [get_nets  
clockName]>
```

If the specified clock is the frequency reference for the DTD2 transceiver logic, you can use the backbone clock trace for routing without negatively affecting performance. You can set the CLOCK_DEDICATED_ROUTE_BACKBONE property in the xdc file as suggested in the error message. For example:

```
set_property CLOCK_DEDICATED_ROUTE_BACKBONE  
[get_nets ident_coreinst/IICE_INST/dtd2_ref_clock]
```

Configuring Resets for DTD2

You must properly configure the reset for the DTD2 IP in the user FPGA before running debug, because link training uses the HAPS global reset signal as the reset signal for the DTD2 IP. The reset signal must be properly synchronized to a sample clock domain and distributed to all the FPGAs. See the *Multi-FPGA Reset Synchronization* section in the downloadable *FPGA-Based Prototyping Methodology Manual* (www.synopsys.com/fpmm) for details.



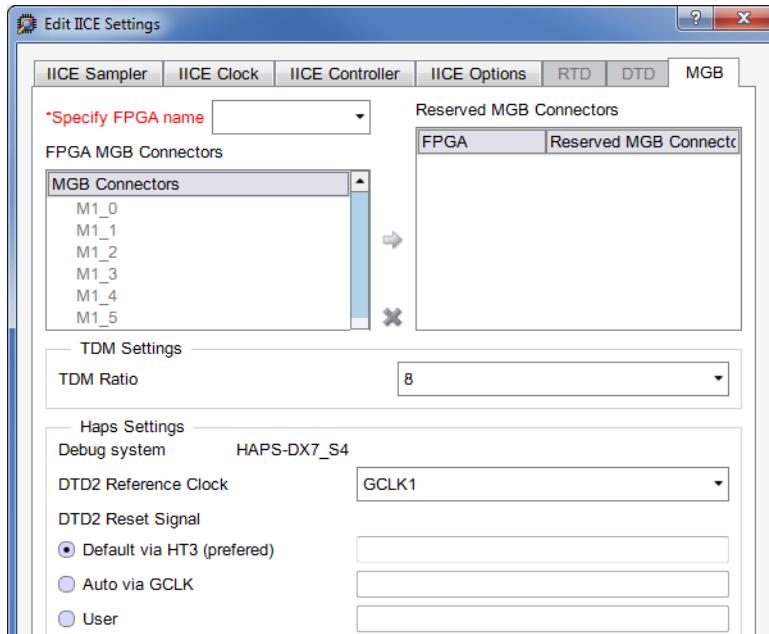
The instrumentor automates reset insertion and inserts a controlled reset line to handle a graceful reset sequence of the DTD2 logic. You do not have to instantiate a reset block or specify a reset driver. The master reset source is based on a pulse generated from the System IP. You can distribute the reset among the FPGAs using either HT3 cables or an available GCLK line.

1. Specify the kind of reset distribution:

| Reset Distribution | iice sampler Command Option |
|-----------------------|--|
| HapsTrak 3 connectors | <code>iice sampler -iice value -DTD2_reset_type {AUTO} [-DTD2_reset board.resetGeneratingFPGA]</code> |
| Global clock resource | <code>iice sampler -iice value -DTD2_reset_type {GCLK} [-DTD2_reset board.GCLKforResetDistribution]</code> |

With the AUTO setting, DTD2 uses the existing connectivity of the system and distributes the reset through the HapsTrak 3 connectors. With GCLK set, DTD2 uses a global routing resource to distribute the reset. The master reset is sourced from the System IP CAPIM.

You can also specify the kind of reset distribution from the GUI (DTD2 Reset Signal option on the MGB tab of the IICE settings dialog box):



2. Establish link training between the user FPGA transciever link and the transciever links on the debug hub.
 - In the exported runtime directory, locate the `mfhdtd_link.tcl` script.
 - Run the script from Confpro. For example: `confprosh mfhdtd_link.tcl`
3. Start the debugger after running the script.

The debugger automatically checks the link status for each run. It initiates link training the first time debug is run. On subsequent runs, it only reports link status in the runtime window, and with the LEDs on the HAPS-DX7 system.

Allocating Locations for PLLs and MMCMs

The tool automatically specifies locations for the PLL and MMCM used by DTD DDR3 and locks them. The default locations for the supported HT3 DDR3 daughter cards are as follows:

| DDR3 Location | Default PLL and MMCM Location |
|----------------------|--|
| J1-J2-J3 | define_attribute [get_cells -hier { *snps_ident_clock_gen_0.PLL_ADV_inst }] LOC PLLE2_ADV_X1Y10 define_attribute [get_cells -hier { *snps_ident_clock_gen_0.MMCM_ADV_inst }] LOC MMCME2_ADV_X1Y10 |
| J4-J5-J6 | define_attribute [get_cells -hier { *snps_ident_clock_gen_0.PLL_ADV_inst }] LOC PLLE2_ADV_X1Y7 define_attribute [get_cells -hier { *snps_ident_clock_gen_0.MMCM_ADV_inst }] LOC MMCME2_ADV_X1Y7 |
| J7-J8-J9 | define_attribute [get_cells -hier { *snps_ident_clock_gen_0.PLL_ADV_inst }] LOC PLLE2_ADV_X1Y4 define_attribute [get_cells -hier { *snps_ident_clock_gen_0.MMCM_ADV_inst }] LOC MMCME2_ADV_X1Y4 |
| J10-J11-J12 | define_attribute [get_cells -hier { *snps_ident_clock_gen_0.PLL_ADV_inst }] LOC PLLE2_ADV_X1Y1 define_attribute [get_cells -hier { *snps_ident_clock_gen_0.MMCM_ADV_inst }] LOC MMCME2_ADV_X1Y1 |

| | |
|-------------|--|
| J1-J2-J3 | define_attribute [get_cells -hier { *snps_ident_clock_gen_0.PLL_ADV_inst }] LOC PLLE2_ADV_X1Y10 define_attribute [get_cells -hier { *snps_ident_clock_gen_0.MMCM_ADV_inst }] LOC MMCME2_ADV_X1Y10 |
| J4-J5-J6 | define_attribute [get_cells -hier { *snps_ident_clock_gen_0.PLL_ADV_inst }] LOC PLLE2_ADV_X1Y7 define_attribute [get_cells -hier { *snps_ident_clock_gen_0.MMCM_ADV_inst }] LOC MMCME2_ADV_X1Y7 |
| J7-J8-J9 | define_attribute [get_cells -hier { *snps_ident_clock_gen_0.PLL_ADV_inst }] LOC PLLE2_ADV_X1Y4 define_attribute [get_cells -hier { *snps_ident_clock_gen_0.MMCM_ADV_inst }] LOC MMCME2_ADV_X1Y4 |
| J10-J11-J12 | define_attribute [get_cells -hier { *snps_ident_clock_gen_0.PLL_ADV_inst }] LOC PLLE2_ADV_X1Y1 define_attribute [get_cells -hier { *snps_ident_clock_gen_0.MMCM_ADV_inst }] LOC MMCME2_ADV_X1Y1 |

If a default location conflicts with design resource allocation, you can reassign PLLs and MMCMs to other locations, according to this table:

| DDR3 Location | Alternative PLL and MMCM Locations |
|----------------------|---|
| J1-J2-J3 | PLLE2_ADV_X1Y11, MMCME2_ADV_X1Y11 PLLE2_ADV_X1Y10, MMCME2_ADV_X1Y10 PLLE2_ADV_X1Y9, MMCME2_ADV_X1Y9 |
| J4-J5-J6 | PLLE2_ADV_X1Y8, MMCME2_ADV_X1Y8 PLLE2_ADV_X1Y7, MMCME2_ADV_X1Y7 PLLE2_ADV_X1Y6, MMCME2_ADV_X1Y6 |
| J7-J8-J9 | PLLE2_ADV_X1Y5, MMCME2_ADV_X1Y5 PLLE2_ADV_X1Y4, MMCME2_ADV_X1Y4 PLLE2_ADV_X1Y3, MMCME2_ADV_X1Y3 |
| J10-J11-J12 | PLLE2_ADV_X1Y2, MMCME2_ADV_X1Y2 PLLE2_ADV_X1Y1, MMCME2_ADV_X1Y1 PLLE2_ADV_X1Y0, MMCME2_ADV_X1Y0 |

Setting up Hardware for DTD2 with HAPS-70

DTD2 is tied to the hardware, so there are some hardware setup and other issues to take into account for DTD2 with HAPS-70 systems. For details, see the following:

- [Connecting the Systems for DTD2](#), on page 731
- [Using SATA Cable MGT Links](#), on page 734
- [Reserving MGT Links](#), on page 736
- [Setting up Fixed Debug Cable Connections](#), on page 737
- [Chaining HAPS-70 Systems for Debug](#), on page 738

For information about running DTD2, see [Using a Debug Hub with SATA \(DTD2\)](#), on page 716.

Connecting the HAPS-70 and HAPS-DX7 Systems for DTD2

This section contains some details about connecting the HAPS-70 and HAPS-DX7 systems for multi-FPGA DTD2.

- [Connecting the Systems for DTD2](#), on page 731
- [Hardware Components for DTD2 \(with HAPS-70\)](#), on page 732
- [Using SATA Cable MGT Links](#), on page 734

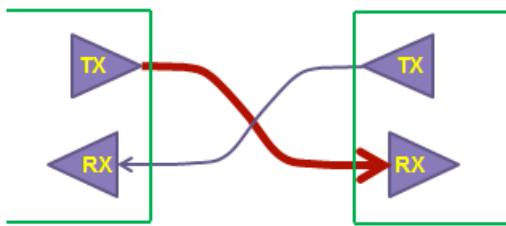
Connecting the Systems for DTD2

1. Make sure you have the appropriate hardware and connect the hardware.

Check the hardware requirements in [Hardware Components for DTD2 \(with HAPS-70\)](#), on page 732 and the connection example in [Hardware Connection Example for DTD2](#), on page 733.

2. Make sure to connect the SATA cables correctly.

See [Using SATA Cable MGT Links](#), on page 734 for more information about connecting the SATA cables.



3. Use the SMB coax cable to connect the HAPS-DX7 and HAPS-70 systems.

This cable provides the HAPS-DX7 debug hub with the required synchronization clock. If you are chaining multiple HAPS-70 systems, make sure that the HAPS-DX7 system is at the end of the UMRBus chain.

Hardware Components for DTD2 (with HAPS-70)

To run multi-FPGA HAPS DTD with the HAPS-70 system, you require the appropriate hardware; the components are listed in the table below. For a video showing the connections and setup, see SolvNet article 2155494, *Multi-FPGA HAPS DTD Component Assembly*.

| 4-FPGA DTD | 8-FPGA DTD |
|---|--|
| HAPS DTD 4-FPGA System Kit (4-FPGA kit for debug, including a HAPS-DX7 system and 8 one-meter SATA cables) | HAPS DTD 4-FPGA System |
| HAPS DTD 4-FPGA Kit (Cables and riser MGB for debug) | HAPS DTD 4-FPGA Add-on Pack (Additional 4 FPGAs) |
| | HAPS DTD 4-FPGA Kit (Additional cards for 8-FPGA setup) |

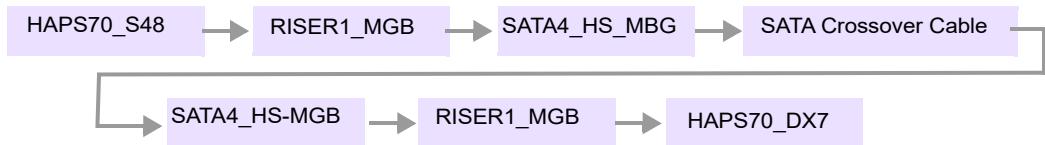
Here are the details of the connection hardware required for multi-FPGA DTD with the HAPS-70 system:

- 1 SATA card and 1 riser card per FPGA
For an eight-FPGA system, this means you require eight SATA and riser card pairs.
- 2 SATA cards and 1 riser card for the HAPS-DX7 system, which is plugged in MGB1

- 8 special SATA cross-over cables, required for proper cross-over and orientation connections for the hardware debug configuration. The cables are 1 m in length. Contact Synopsys if you require a different length.

Hardware Connection Example for DTD2

This example shows a setup with the HAPS systems, accessories, and connections needed to use a HAPS system as a debug board for deep trace debug:



For pinout and connection information for the systems, daughter boards, and cables, refer to the corresponding documentation, which is available from the HAPS SupportNet website on SolvNet (<https://solvnet.synopsys.com>).

| Hardware | Description | | | | | | | | | | | | | | | | |
|----------------------|---|----|----|----------|----------|-------|-------|-------|-------|----------|----------|-------|-------|-------|-------|----------|----------|
| HAPS70_S48 | HAPS system. <i>HAPS -70 S48 Reference Manual</i> | | | | | | | | | | | | | | | | |
| RISER1_MGB | MGB card that divides an MGB connector into two equal halves. <i>HAPS MGB Interface Cards Reference Manual</i> | | | | | | | | | | | | | | | | |
| SATA4_HS_MGB | MGB interface card with 4 serial ATA connection channels. <i>HAPS MGB Interface Cards Reference Manual</i> | | | | | | | | | | | | | | | | |
| SATA Crossover Cable | Serial link for point-to-point connection between devices. The figure shows the crossover wiring: | | | | | | | | | | | | | | | | |
| | <table border="1"> <thead> <tr> <th>P1</th> <th>P2</th> </tr> </thead> <tbody> <tr> <td>Ground 1</td> <td>7 Ground</td> </tr> <tr> <td>A + 2</td> <td>6 B +</td> </tr> <tr> <td>A - 3</td> <td>5 B -</td> </tr> <tr> <td>Ground 4</td> <td>4 Ground</td> </tr> <tr> <td>B - 5</td> <td>3 A -</td> </tr> <tr> <td>B + 6</td> <td>2 A +</td> </tr> <tr> <td>Ground 7</td> <td>1 Ground</td> </tr> </tbody> </table> | P1 | P2 | Ground 1 | 7 Ground | A + 2 | 6 B + | A - 3 | 5 B - | Ground 4 | 4 Ground | B - 5 | 3 A - | B + 6 | 2 A + | Ground 7 | 1 Ground |
| P1 | P2 | | | | | | | | | | | | | | | | |
| Ground 1 | 7 Ground | | | | | | | | | | | | | | | | |
| A + 2 | 6 B + | | | | | | | | | | | | | | | | |
| A - 3 | 5 B - | | | | | | | | | | | | | | | | |
| Ground 4 | 4 Ground | | | | | | | | | | | | | | | | |
| B - 5 | 3 A - | | | | | | | | | | | | | | | | |
| B + 6 | 2 A + | | | | | | | | | | | | | | | | |
| Ground 7 | 1 Ground | | | | | | | | | | | | | | | | |
| HAPS70_DX7 | HAPS system. <i>HAPS Developer eXpress System Hardware Reference Manual</i> | | | | | | | | | | | | | | | | |

Using SATA Cable MGT Links

For each debug session, you can have up to 8 MGT links or SATA cables. Allocate the MGT links according to your specific debug needs. Some allocation examples are shown below; this is not an exhaustive list. For a video illustrating the connections and setup, see SolvNet article 2155494, *Multi-FPGA HAPS DTD Component Assembly*.

| HAPS-70 | | | | HAPS-DX7 |
|---------|--------|--------|--------|----------|
| FPGA A | FPGA B | FPGA C | FPGA D | |
| 4 | 4 | 0 | 0 | 8 |
| 0 | 4 | 0 | 4 | 8 |
| 2 | 2 | 2 | 2 | 8 |
| 8 | 0 | 0 | 0 | 8 |
| 2 | 1 | 4 | 1 | 8 |
| 2 | 2 | 4 | 0 | 8 |
| 2 | 2 | 0 | 0 | 4 |
| 0 | 0 | 0 | 4 | 4 |

Note that the configuration you use affects the maximum sample frequency, as described in [Working with Maximum Sample Frequency for DTD2, on page 724](#).

Each HAPS-70 FPGA has the following MGT links, that can potentially be used for MGHD TD debug:

MGB1.J2.X2, MGB1.J2.X3, MGB1.J2.X4, MGB1.J2.X5
MGB1.J3.X2, MGB1.J3.X3, MGB1.J3.X4, MGB1.J3.X5
MGB2.J2.X2, MGB2.J2.X3, MGB2.J2.X4, MGB2.J2.X5
MGB2.J3.X2, MGB2.J3.X3, MGB2.J3.X4, MGB2.J3.X5

The instrumentor selects the maximum number of MGT links possible for debug from the list so as to maximize sample frequency. It does not use links that are marked as reserved. The actual number of links used depends on debug link availability on the debug hub HAPS-DX7 and the maximum of the debug links available for other FPGAs.

Example 1: MGT Link Allocation with no Duplexing

In this example, there are four FPGAs to be instrumented, and each FPGA has all the MGT links available for debug.

The tool instruments the signals and allocates debug links as follows, because this configuration achieves a maximum sample frequency of 70 MHz:

| | Instrumented Signals | Debug Links |
|--------|-----------------------------|--------------------|
| FPGA A | 30 | 1 |
| FPGA B | 64 | 2 |
| FPGA C | 120 | 4 |
| FPGA D | 32 | 1 |

The MGT link duplexing factor is 1.

Example 2: MGT Link Allocation with Duplexing = 2

Just as in Example 1, this example has four FPGAs to be instrumented, and each FPGA has all the MGT links available for debug.

The tool instruments the signals and allocates debug links as follows, because this configuration achieves a maximum sample frequency of 35 MHz:

| | Instrumented Signals | Debug Links |
|--------|-----------------------------|--------------------|
| FPGA A | 60 | 1 |
| FPGA B | 64 | 1 |
| FPGA C | 120 | 2 |
| FPGA D | 32 | 1 |

The MGT link duplexing factor is 2. Although the other FPGAs do not need to be duplexed, the limiting factor is FPGA C, whose 120 signals must be duplexed to fit the two available links. To support the 70 MHz sample frequency (duplexing factor of 1) in Example 1 would require nine debug links. This is currently not supported.

Example 3: MGT Link Allocation with Fewer Available Links

In this example, there are four FPGAs to be instrumented, but each FPGA has only two MGT links available for debug.

The tool instruments the signals and allocates debug links as follows, because this configuration achieves a maximum sample frequency of 35 MHz:

| | Instrumented Signals | Debug Links |
|--------|----------------------|-------------|
| FPGA A | 30 | 1 |
| FPGA B | 64 | 1 |
| FPGA C | 120 | 2 |
| FPGA D | 32 | 1 |

The MGT link duplexing factor is 2. The limiting factors are the 120 signals on FPGA C that must be duplexed to fit the limited number of available links.

Using the HapsTrak 3 Cable

To properly represent reset conditions for a design partitioned into multiple FPGAs, the reset signal is synchronized to the design clock domain. You can use a HapsTrak 3 cable to distribute the reset signal between the HAPS-70 user FPGAs and ensure the proper reset conditions.

For a video showing the connections and setup, see SolvNet article 2155494, *Multi-FPGA HAPS DTD Component Assembly*. For a detailed discussion of replicating and synchronizing resets, see the *Multi-FPGA Reset Synchronization* section in the *FPGA-Based Prototyping Methodology Manual* (www.synopsys.com/fpmm).

The HapsTrak 3 cable is not included in HAPS DTD 4-FPGA System Kit or the HAPS DTD 4-FPGA Add-On Pack.

Reserving MGT Links

To mark an MGT link reserved, use the `iice sampler -gtx` command in the `idc` file. For example:

- For a single HAPS-70 system, where all the MTG transceivers are available for debug use, use a command like the one below. The example is for a HAPS-70 S24 system:

```
iice sampler -gtx {mb1.uA {} mb1.uB {}}
```

- For two chained HAPS-70 systems, use a command like the iice sampler -gtx command shown below. The example is for two chained HAPS-70 S24 systems with the following MGT transceivers available:

| System 1 | System 2 |
|-----------------|-----------------|
| FPGA A: MGB1 | FPGA A: MGB2 |
| FPGA B: MGB2 | FPGA B: MGB1 |

```
iice sampler -gtx {\nmb1.uA { MGB2.J2.X2 MGB2.J2.X3 MGB2.J2.X4 MGB2.J2.X5 MGB2.J3.X2\nMGB2.J3.X3 MGB2.J3.X4 MGB2.J3.X5 } \\\nmb1.uB { MGB1.J2.X2 MGB1.J2.X3 MGB1.J2.X4 MGB1.J2.X5 MGB1.J3.X2\nMGB1.J3.X3 MGB1.J3.X4 MGB1.J3.X5 } \\\nmb2.uA { MGB1.J2.X2 MGB1.J2.X3 MGB1.J2.X4 MGB1.J2.X5 MGB1.J3.X2\nMGB1.J3.X3 MGB1.J3.X4 MGB1.J3.X5 } \\\nmb2.uB { MGB2.J2.X2 MGB2.J2.X3 MGB2.J2.X4 MGB2.J2.X5 MGB2.J3.X2\nMGB2.J3.X3 MGB2.J3.X4 MGB2.J3.X5 } \\\n}
```

Setting up Fixed Debug Cable Connections

To avoid having to modify cable connections when the number of signals being instrumented changes, you can set up a fixed DTD2 cable setup. To do so, use the iice sampler -gtx command to reserve all MGT links except the ones to which the debug cable is connected.

For example, take a setup where a HAPS-70 S24 system has the following debug links connected to the HAPS-DX7 debug hub:

| | |
|--------|------------------------|
| FPGA A | MGB1.J2.X2, MGB1.J2.X5 |
| FPGA B | MGB2.J3.X3, MGB2.J3.X4 |

To ensure that the tool always selects the debug links that already have the cable connected, add the following command to the idc file :

```
iice sampler -gtx {
mb1.uA { MGB1.J2.X3 MGB1.J2.X4 MGB1.J3.X2 MGB1.J3.X3 MGB1.J3.X4
MGB1.J3.X5 MGB2.J2.X2 MGB2.J2.X3 MGB2.J2.X4 MGB2.J2.X5 MGB2.J3.X2
MGB2.J3.X3 MGB2.J3.X4 MGB2.J3.X5 } \
mb1.uB { MGB1.J2.X2 MGB1.J2.X3 MGB1.J2.X4 MGB1.J2.X5 MGB1.J3.X2
MGB1.J3.X3 MGB1.J3.X4 MGB1.J3.X5 MGB2.J2.X2 MGB2.J2.X3 MGB2.J2.X4
MGB2.J2.X5 MGB2.J3.X2 MGB2.J3.X5 } \
}
```

Note that as this method fixes the number of debug links, the maximum sample frequency may go down if more signals are instrumented.

Chaining HAPS-70 Systems for Debug

You can chain up to three HAPS-70 systems together in a single debug session.

1. Chain up to three HAPS-70 systems together.
2. Make sure the HAPS-DX debug hub is at the end of the UMRBus chain.
3. Use CLK_LEFT and CLK_RIGHT to distribute the global clock between the three systems.

One of the systems will use different GCLKs than the other two. Take an example where systems FB1 and FB2 use GCLK3. Where FB2 gets global clocks from FB1 through the CLK_LEFT connector, FB3 gets clocks from FB1 via CLK_RIGHT. GCLK3 for FB1 and FB2 shows up in FB3 as GCLK9. Currently, the run system_generate command does not automatically take this into account.

4. Add a A TCL script segment to your tool TCL script to account for the clock differences.

See [Example: Tcl Script Segment to Account for Different GCLKs](#), on page 738.

Example: Tcl Script Segment to Account for Different GCLKs

```
# Change GCLK3 to GCLK9 on mb3_uB pinloc
set origfile devices/mb3_uB/syn_dics_mb3_uB.fdc
set savefile devices/mb3_uB/syn_dics_mb3_uB_ORIG.fdc
if [file exists $origfile] {
    if {! [file exists $savefile]} {
        file copy $origfile $savefile
```

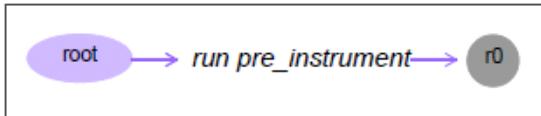
```
        }
        set IH1011 [open $savefile]
        set OH1011 [open $origfile w]
        while {[gets $IH1011 line] > -1} {
            if [regexp {xc_loc} $line] {
                regsub {{AB40}} $line {{C27}} line
                regsub {{AA40}} $line {{C28}} line
            }
            puts $OH1011 $line
        }
        close $IH1011
        close $OH1011
    }
```

Using a Debug Hub with QSFP Connectors

With the DTD3 deep trace debug scheme, you use QSFP (Quad Small Form-factor Pluggable) transceiver connectors and a HAPS-DX7 system as a debug hub to run multi-FPGA debug. The QSFP connectors support up to eight high-speed serial links over two QSFP+ cables. You can use this external debug memory setup with HAPS-80 systems.

1. Instrument before partitioning.

- Specify the `run pre_instrument` command to create a database state for instrumentation:



- Run the `edit idc` command to start the instrumentor.

Specify the `-mode gui` argument to work in the graphical interface. In the GUI, click the Edit IICE Settings icon, and set the options in the GUI. For example, set Buffer type to Multi-FPGA DTD Module on the IICE Sampler tab of the dialog box.

- Specify the MGB connectors to use for debug.
- Set the sample depth.
- Specify sample, instrument signals, and configure trigger engine.

2. Specify MGT links and memory settings.

- Add the debug memory hub:

```
iice mgb -iice {myiice} -add_hub {myDebugMem}
```

You can use a total of four QSFP links per hub to support up to four FPGAs. You can also chain two hubs together.

- Reserve user links, as needed. For example:

```
iice mgb -iice {my_iice} -mgb_reserved {mb.uC.MGB1.J2A}
```

- Specify links for debug memory:

```
iice mgb -iice {IICE_0} -connect {FB1.uA.MGB2.J2A} {hubA.MGB1.J2A}  
iice mgb -iice {IICE_0} -connect {FB1.uD.MGB1.J2A} {hubA.MGB1.J2B}
```

- Debug link and speed settings:

```
iice mgb -iice {IICE_0} -num_ddr3_locations {2}
iice mgb -iice {IICE_0} -mgb_max_links {16}
iice mgb -iice {IICE_0} -mgb_link_speed {6}
```

- Set debug memory settings:

```
iice mgf -iice {myiice_name} -num_ddr3_locations {2}
```

3. Set clocks and resets.

- | | |
|---------|--|
| HAPS-70 | <ul style="list-style-type: none"> • Set the DTD reference clock to a 100 MHz GCLK driven by the HAPS system (e.g. GCLK3). • Synchronize reset distribution to multiple FPGAs using HT3 cables. • Set TSS and Confpro configuration commands to match this setup. |
| HAPS-80 | <p>Set the DTD reference clock to a 150 MHz GCLK driven by the HAPS-DX7 hub.</p> <ul style="list-style-type: none"> • Synchronize reset distribution to multiple FPGAs using HT3 cables. • Set TSS and Confpro configuration commands to match this setup. |

IDC File Commands:

```
iice sampler -iice {IICE_0} -dtd_clock {GCLK3}
iice sampler -iice {IICE_0} -dtd_reset_type {AUTO}
```

TSS:

```
board_system_configure -clock {FB1.GCLK3} pll
```

Confpro:

```
cfg_clock_set_frequency$h FB1 PLL1_3 100MHz
cfg_clock_set_select$h FB1 GCLK3 PLL
```

DTD Hub (HAPS-DX7 side):

```
cfg_clock_set_input$h FB2 PLL1 PLLIN 100MHz
cfg_clock_set_select$h FB2 GCLK1 PLL
```

4. Run debug on the design. You get multi-FPGA debug visibility. Connect the QSFP board and cables for debug requirements:

- Connect FPGAs to the debug hub:

| Connect... | To... |
|-----------------|--------------------|
| FPGA A MGB1 J2A | Debug Hub MGB1 J2A |
| FPGA B MGB1 J2A | Debug Hub MGB1 J2B |
| FPGA C MGB1 J2A | Debug Hub MGB3 J2A |
| FPGA D MGB1 J2A | Debug Hub MGB3 J2B |

- Connect the SMB coax cable for the reference clock, by connecting the HAPS-80 EXTIN3 to PLL OUT1 on the debug hub.

5. Run debug.

- Launch ProtoCompiler runtime and run link training.

Using BRAM for Debugging

You can use the on-board BRAM blocks to store sample data for debugging operations. BRAM-based debug offers many advantages. It is a platform-independent, fast, method that does not consume I/O resources. It leverages existing system resources, so it does not require additional hardware resources. It uses existing TSS connectivity so no major TSS changes are required. The performance limits and width are determined by the FPGA resources.

You can use it for single-FPGA or multi-FPGA debug for HAPS-70, HAPS-80, and mixed systems. You can use it for any sampling frequency, but it is best suited to high-frequency debugging.

1. Instrument the design.

- For single-FPGA designs, instrument the design at the pre-instrument state (RTL-based instrumentation) or after compile (compiled database instrumentation).
- For multi-FPGA designs, instrument the design at the top level at the pre-instrument state (RTL-based instrumentation) or the pre-partition state (compiled database instrumentation).
- Set the buffer type to internal memory:

```
iice sampler iice internal_memory
```

- Set simple triggering only.

2. Run through the partitioning or synthesis implementation flow as usual.

The tool uses four pins per slave IICE. It automatically inserts a cross-trigger network

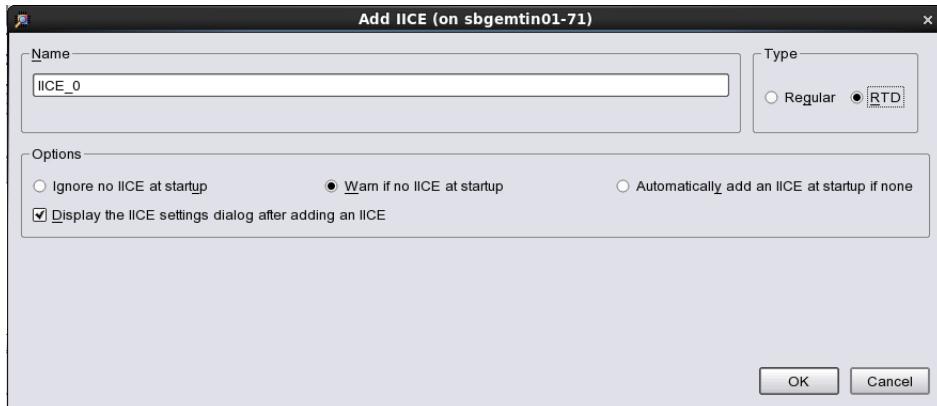
3. Start ProtoCompiler runtime and run debug.

The tool uses distributed RAM blocks to store sample data. You can view the results in a single waveform view (FSDB, VCD). It generates a balanced network where waveforms do not need to be post-processed.

Using Real-time Debugging

Real-time debugging (RTD) provides scope or logic analyzer access to instrumented signals directly through a Mictor board interface connector installed on the HAPS system.

1. Enable real-time debugging by defining a special IICE from the GUI or the TCL shell.
 - To specify the IICE from the user interface, click the Add IICE icon to display the following dialog box. Select RTD as the type of IICE and click OK.



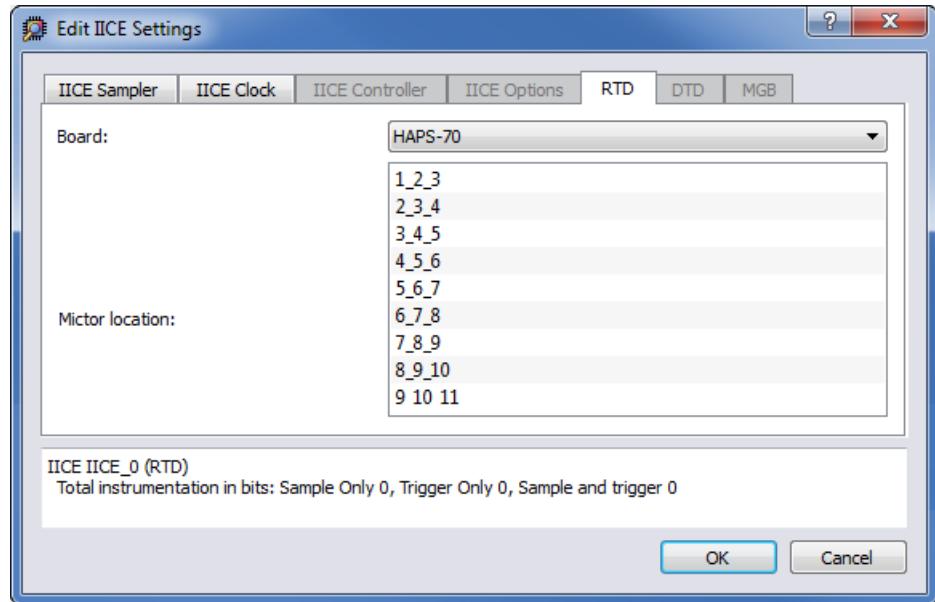
- To define the IICE from the TCL Script shell, enter the `iice new` command, and specify rtd as the type of IICE:

```
iice new [iiceID] -type rtd
```

In the command syntax, `iiceID` is the name of the new IICE and, if omitted, defaults to an incremental number (for example, IICE 2).

Either of these methods creates a new, real-time IICE for the design with all of the signals “not instrumented.” In GUI mode, it also opens a dialog box where you can configure the IICE settings.

2. Configure the IICE.
 - On the RTD tab, select the HAPS system.
 - Specify the HapsTrak® 3 connector locations for the Mictor board by clicking on the appropriate set of connectors.



- Click OK.
- 3. Instrument the RTD signals.

Identify signal watchpoints and breakpoints as with normal instrumentation. You can only set watchpoints on RTD signals to Sample only.

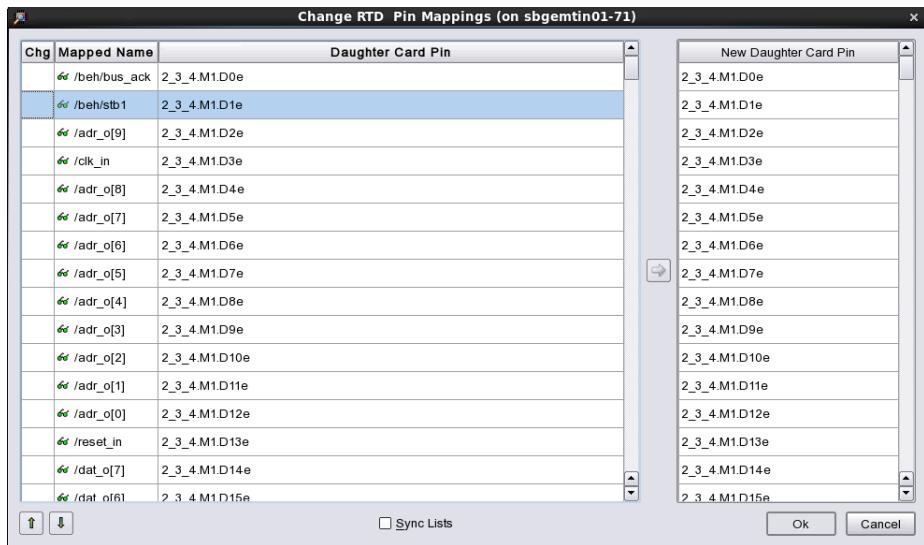
```

289 & bus ack <= & rom_ack and & raml_ack and & ram2_ack;
290 & ack1_t <= & grant1;
291 & ack2_t <= & grant2;
292 & bus_st <= when ( & grant1 = '1' ) else & stb2;
293 & bus_we <= when ( & grant1 = '1' ) else & we2;
294
295 -- bus d
296 & bus_da <= Add mux group...
297 & bus_dat <= Copy
298
299 & bus_dat_m2s <= & dat1_m2s when ( & grant1 = '1' ) else & dat2_m2s;
300

```

- 4. Check the signal assignments.
- Click the Edit RTD IIICE mappings icon in the menu bar (at the top of the instrumentor view. This opens the Change RTD Pin Mappings

dialog box, which displays the Mictor daughter board pin locations to which the watchpointed signals are assigned.



- To change an assignment, highlight the assignment in the left panel (use the Up or Down arrows if necessary). Select the new Mictor pin in the right panel (use the vertical scroll bar if necessary). Click the arrow located between the panels. The New Mictor Pin section on the right displays the new assignment.

Once all the instrumentation is set up you can probe the design when you run the debug phase of the design.

- Run the design through the normal partition, synthesis and place-and-route flow.
- Start the debugger and configure the logic analyzer interface.
 - Set up the hardware and the logic analyzer.
 - Export the design for the runtime executable.
 - Start the runtime executable to run debug.
 - From the debugger GUI, click the RTD icon (). In the dialog box that opens, go to the Logic Analyzer scan tab and specify the type of logic analyzer, the script, and the host name. To automatically assign pods to the Mictor connectors, enable Assign pods automatically to Mictor connectors.

- Click Scan Logic Analyzer. This opens a network connection, retrieves information about the module and pods, and makes additional tabs available in the GUI.
- Go to the Logic Analyzer Properties tab and manually assign Mictor pin groups to modules and pods.
- When you are satisfied with the assignments, go to the Logic Analyzer Submit tab and click Submit.

For details about the options, see [Logic Analyzer Interface Parameters, on page 60](#) in the *Debugger User Guide*.

Running Incremental Debug for ECOs

This instrumentation flow allows Engineering Change Orders (ECOs) to modify existing instrumentation for an IICE. It leverages the Xilinx incremental re-route feature. When ECO-based incremental re-route is used for a new debug implementation, turnaround time is much faster than running a full compile for a few instrumentation changes to the IICE logic.

Incremental debugging for ECOs is currently supported on single FPGA synthesis flow for BRAM and haps100_DTD_builtin buffer types.

1. Complete the initial instrumentation of a design, run regular synthesis flow, followed by Vivado place-and-route and bit file generation, and import the Vivado results to map state using `import vivado` command.
2. Create an incremental idc file.
 - Run `edit idc` or `edit idc <idc_filename.idc> -mode gui` from the mapped state and provide a new name for the incremental IDC file. See the *Command Reference Manual* for syntax details.
3. (Optional) Run the `report debug_visibility` command at the map state. The signal names contained in the IDC (RTL or SRS) are converted into register names required by the Debug ECO and the availability of the signal names are checked and reported. The feature allows you to instrument new signals based on RTL or SRS signal names instead of Xilinx register names.
Even though this is an optional step, it is a good practice to run the `report debug_visibility` command and find if a signal is optimized or renamed.
4. Edit the instrumentation for ECO modifications.

You can edit the existing instrumented signals and also instrument new signals.

- In the GUI, the Instrumentor shows the original instrumentation (described in the original idc file) from the mapped database. Select the module from the Hierarchy View and click the RTL or Instrumentation tabs to see the current instrumentation. The colors of the glasses icons () indicate instrumentation status. Black glasses mark signals that cannot be modified. The image below shows an example of the RTL view in the Instrumentor, for a standard compile HDL database design. In case of a unified compile flow, the RTL tab is not available.

The screenshot shows the Vivado Instrumentor interface. On the left, a tree view displays the 'ROOT (bus_de...)' hierarchy, which includes components like 'beh', 'word_xfe...', 'blk_xfer...', 'arb_inst...', 'ron_inst...', 'ram1_ins...', and 'ram2_ins...'. One node, 'U1 (IBU...', is expanded to show its internal structure. On the right, two tabs are visible: 'RTL (arb.vhd)' and 'Instrumentation'. The 'RTL (arb.vhd)' tab contains the VHDL code for an arbiter:

```

19 -- bus by two bus masters. Both masters have the same priority,
20 -- arbitrator attempts to be "fair" in that if both masters req
21 -- the one which had it last does NOT get it.
22 -- The arbiter is modified to use two clocks to demonstrate cross
23 -- One, clk, is used only by the arbiter and the other, clk_sys
24 -- remaining logic blocks. The clk signal here is of a different
25 -- than clk_sys which is the clk for the rest of the system.
26 -----
27 -- this is another comment
28 library IEEE;
29 use IEEE.std_logic_1164.all;
30 use IEEE.std_logic_arith.all;
31
32 entity arbiter is
33 port (
34   clk : in std_logic;
35   clk_svs : in std_logic;
36   reset : in std_logic;
37   r1 : in std_logic;
38   r2 : in std_logic;
39   grant1 : out std_logic;
40   grant2 : out std_logic
41 );
42 end arbiter;
43
44 architecture beh of arbiter is
45 type states is ( st_idle1, st_idle2, st_grant1, st_grant2);
46 signal curr_state, next_state: states;
47 signal dreq1, dreq2, dreq11, dreq22, dgrant1, dgrant22
48
49 begin
50 process(clk, reset)

```

- Use the Root hierarchy in the Instrumentor, if back annotation is used in the mapped state of the design. The signals are mapped to the HDL signal names. This is applicable for Standard Compile flow. Click the Hierarchical check box in the Instrumentor to list the signals from the whole RTL design, including its sub-hierarchies.
- Use the INCR_FILE hierarchy in the Instrumentor. This is applicable only for the SRS database of a single FPGA. The INCR_FILE has a list of the Vivado EDIF mapped signals of your design, which is available for incremental instrumentation.

Use the following Tcl command in Instrumentor shell window to generate a list of instrumented signals:

```
% hierarchy find -stat instrumented -type signal
```

To quickly filter the signals that you can edit, click the Instrumentor Search icon (🔍) in the upper left of the Instrumentor window to open the Instrumentor Search dialog box. Set the Instrumentation option to

not-instrumented, and click Search. The tool displays a filtered list that only contains the un-instrumented signals you can modify.

Use the following Tcl command in Instrumentor shell window to generate a list of un-instrumented signals:

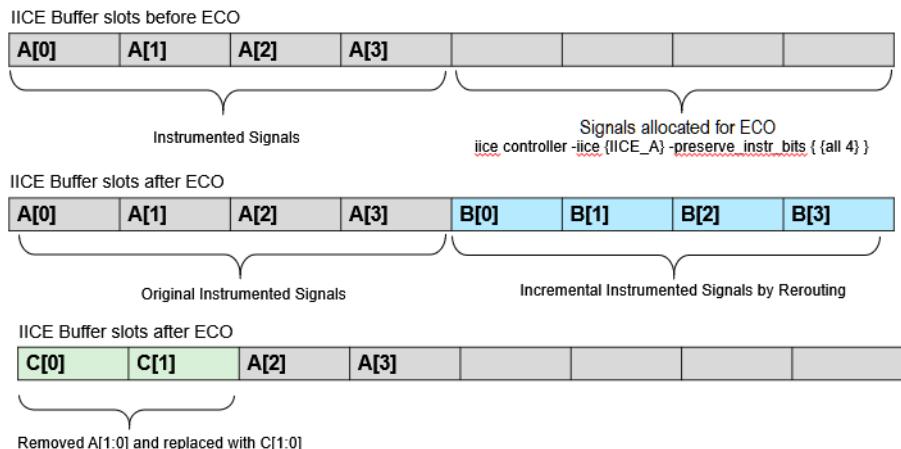
```
% hierarchy find -stat not-instrumented -type signal
```

- Un-instrument the signals, which are expected to be replaced from the initial instrumentation.
- Instrument the new signals.

The total number of signals or bits cannot exceed the original number, so for every new signal or bit added, you must remove an existing signal or bit. To allocate more signal bits than the original instrumentation bits, you can use the following Tcl command during the initial instrumentation step:

```
% iice controller -iice <IIICE_name> -allocate_instr_bits {{all <n>}}
```

where, n represents the number of bits to be allocated for the future instrumentation during ECO. The following image provides an overview of allocate bits scenario:

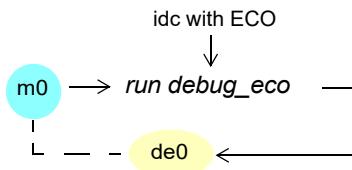


- Save the instrumentation. The tool creates an idc file that only contains the edits, which it uses in conjunction with the original idc file from the mapped state.

- Incorporate the ECO instrumentation into the design with this command:

```
run debug_eco -idc modifiedFile.idc
```

The command uses the specified idc file with the ECO changes, and generates an incremental Vivado script for place and route. The run debug_eco command creates a new state (de0) below the parent map state. This is a terminal state that does not generate any other sub-states. You cannot use run debug_eco command again from the existing debug_eco state. At the de0 state, check the <top_module_name>_identify_script.log log file generated for the incremental instrumentation information. See [run debug_eco, on page 141](#) in the *Command Reference Manual* for syntax details.



- Re-route the design and generate the bit file.
 - From the debug_eco state (de0), run the export vivado command. This command writes out the newly generated Vivado script (debug_eco_vivado.tcl) to the specified location. If you have multiple debug ECO runs and corresponding sub-states, make sure to run the export vivado command from the correct sub-state (de1, de2, and so on).
 - Run launch vivado to run the incremental re-route and generate new bit files. The script runs the Vivado router incrementally. The incrementally re-routed signals information can be found in the vivado.log.
 - From the debug_eco substate, run import vivado to import the place-and-route results into the database.
- Link the debug ECO states to the system generate node using the database apply_state -link_module command after system generate state. The following example shows how to link the debug ECO states of FPGA A and B to the system generate state:

```
database apply_state -link_module FB1_uA
./synthesis_files/FB1_uA/FB1_uA_srs|de0
```

```
database apply_state -link_module FB1_uB  
./synthesis_files/FB1_uB/FB1_uB_srs|de0
```

8. Run export runtime to export the incremental ECO instrumentation for debug.
9. Program the ECO generated bit files on the hardware, and open the debug projects exported after incremental ECO instrumentation in ProtoCompiler Runtime. The incrementally instrumented signals are now available for debug.
10. Debug the design as usual by setting watchpoints and running the debugger.

Limitations to the Debug ECO Flow

Currently, these are some limitations with the debug ECO flow:

- Does not support the following features:
 - Clock groups for multi-clock instrumentation
 - Mux groups with preserve bits
 - Breakpoint instrumentation
 - Debug ECO for partition flow
- IICE buffers must use BRAM or haps100_DTD_builtin buffers

Using Global State Visibility (GSV)

Global State Visibility (GSV) provides a snapshot of all registers in a configured design at a particular clock cycle, so it offers the advantage of full visibility. It does not require instrumentation and is instantly available. GSV correlates the low-level signal names back to the original RTL names. It makes the entire register chain visible for debug, by leveraging the Xilinx register readback capability, which reads the current states of all internal registers within a Xilinx FPGA at a particular clock cycle.

You can use the information for hardware debugging and functional verification. You can run GSV by itself or in combination with other debug functionality for single-FPGA designs, single-FPGA partitions, or multi-FPGA designs. You can run GSV in asynchronous mode for semi-static analysis, or in synchronous mode for system-wide state correlation and clock control.

There are two GSV modes: legacy GSV and enhanced GSV (eGSV). With eGSV mode, there is a significant improvement in downloading speed of the GSV data from the target device. To enable eGSV mode, set the `prepare_readback` option to 2. This mode is implemented only when the design is instrumented but there will not be any change in the GSV use model described in the following sub-sections.

Note that the term GSV is used broadly to describe both GSV and eGSV, as the use models are similar. However, the term eGSV is used when something only applies to eGSV.

See these topics for details:

- [Asynchronous and Synchronous GSV](#), on page 754
- [Using Asynchronous GSV for a Single-FPGA Design](#), on page 757
- [Use Model Example: Synchronous GSV for a Single-FPGA Design](#), on page 760
- [Use Model Example: Synchronous GSV with a Multi-FPGA Design](#), on page 762
- [Instantiating and Configuring Clock Control Modules for GSV](#), on page 764
- [Memory GSV](#), on page 768
- [Example: Implementing Memory GSV for a Single-FPGA Design](#), on page 768

- [Setting up the Hardware for GSV](#), on page 772
- [Running GSV with Clock Control](#), on page 773

Asynchronous and Synchronous GSV

GSV can be asynchronous or synchronous. Use asynchronous mode for analysis of static or semi-static signals, or synchronous mode for system-wide state correlation and clock control.

Synchronous and Asynchronous GSV Comparison Table

The table below compares the features available in asynchronous and synchronous GSV:

| Feature | Synchronous GSV | Asynchronous GSV |
|---|---|---|
| Time-synchronized register values within and across FPGAs | CCM (clock control module) guarantees synchronous readback of all registers in and across FPGA | Register values may not be synchronized |
| Clock stepping | CCM provides stepping | Stepping is NOT possible |
| Trigger support | Yes | No |
| External hardware | <ul style="list-style-type: none"> Hardware required for fine grain control mode, for arming, releasing, stepping. Trigger out with distributed IICE None required for asynchronous CAPIM control which is triggered from the host system command line. | Not required |
| Usage scenarios | <ul style="list-style-type: none"> System hang diagnostics Dynamic system activity (limited window) | <ul style="list-style-type: none"> System hang diagnostics Semi-static signals/discrete signals |

Descriptions of Synchronous and Asynchronous GSV

The following provide more details about the differences between the two modes:

- Asynchronous GSV

Asynchronous GSV represents a snapshot of the register states in a running design, with the captured asynchronously to the design clock. All register data within the design are loaded into a parallel rank register and shifted out. However, as the design clock is still running, there is no guarantee that the captured data are aligned to the same clock cycle. In asynchronous mode, the state of the system is unknown. Asynchronous GSV is best suited for hang conditions, where you are looking at static or semi-static signals, like control signals, to get an idea of where to look for your next instrumentation cycle. For example, you can use this mode when programming an IP, to confirm the state of the IP.

See [Using Asynchronous GSV for a Single-FPGA Design, on page 757](#) and [GSV Limitations](#), on page 757 for information about using this mode.

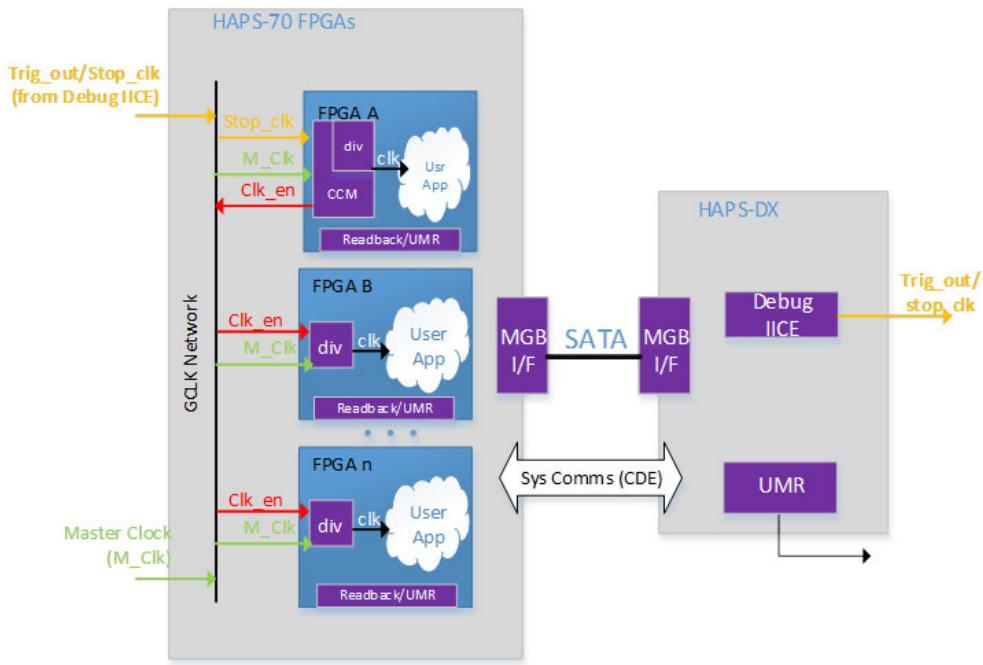
- Synchronous GSV

Synchronous GSV correlates the entire system and aligns all the registers. It is intended for use in a hung processor scenario, where there is no DTD visibility. It provides a snapshot of register values while the design clock domain is stopped. The drawbacks to synchronous GSV are the need to stop clocks and zero depth.

Synchronous GSV requires a clock control module (CCM) to be instantiated. The CCM is inserted at the base of the clock tree for the design. It takes the master clock source as input and forwards the clock to the rest of the design. When an enable signal is asserted, the CCM gates the clock and facilitates the downloading of captured registers off the device.

You can program the CCM with a command sequence, with options to step the clock for a specified number of cycles after the CCM module has been triggered.

There are two ways to control CCM assertion: direct assertion using an input port to the CCM, or through a command from the debugger shell. The following diagram shows a HAPS system used as a deep trace debug (DTD) buffer, and also as the mechanism to assert the control line to the clock enable module.



In the synchronous use model, the DTD debug IICE monitors signals under test at system speed. You set a trigger event, and when this occurs, the debug IICE is also configured to assert a trigger out signal. The trigger out signal is routed back to the user FPGAs to assert the CCM in order to start the GSV capture sequence.

Further details about using synchronous GSV are described in subsequent sections:

- [Use Model Example: Synchronous GSV for a Single-FPGA Design](#), on page 760
- [Use Model Example: Synchronous GSV with a Multi-FPGA Design](#), on page 762
- [Instantiating and Configuring Clock Control Modules for GSV](#), on page 764
- [Setting up the Hardware for GSV](#), on page 772
- [Running GSV with Clock Control](#), on page 773

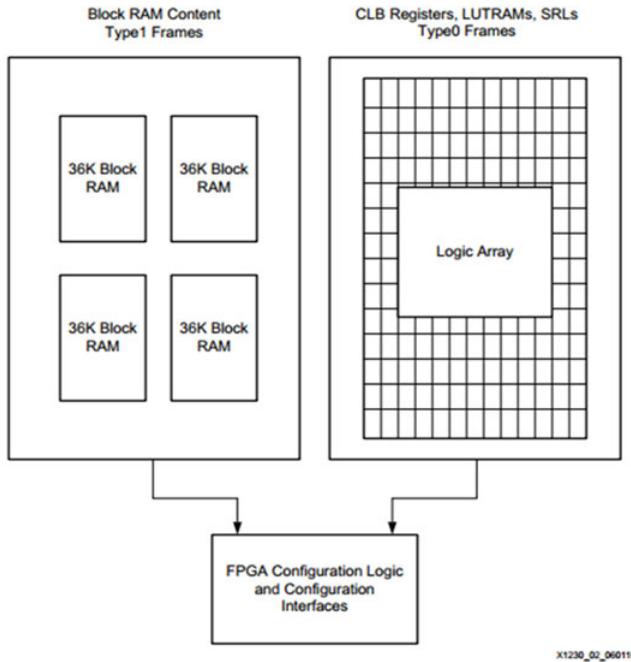
GSV Limitations

Currently, there are some limitations:

- The eGSV flow is applicable for HAPS-80 technology only. For mixed HAPS system, legacy GSV must be adapted.
- The GSV capability uses GPIO and HT3 pins that are usually configured as user design pins, so there could be place-and-route conflicts if you do not set the readback option.
- Synchronous GSV limitations:
 - Clocks generated by the clock control module are synchronous to one another.
 - The phase and frequency from the clock control module can be controlled independently for each clock. Controlled clocks have a duty cycle of 50%.
 - You cannot independently stop an individual clock from the clock control module.
 - GSV is not supported for HAPS-DX flow.
- Use the latest version of `haps_controlled_clocks.v`; do not use older versions saved in local areas.

Using Asynchronous GSV for a Single-FPGA Design

Asynchronous GSV represents a snapshot of the register states in a running design. It is a simple snapshot with no data alignment between signals. It is intended for use with static or semi-static signals.



The following procedure describes how to use asynchronous GSV for a single, unpartitioned FPGA.

1. Set the compile option for GSV or eGSV:

GSV: `option set prepare_readback 1`

eGSV: `option set prepare_readback 2`

You must set this option to generate the logic location (.ll) file later, during bit file creation.

2. Run through the other stages in the flow as usual: compile, map, placement and routing.

When the bit file is created, Vivado generates a .ll file, which maps the register information from the FPGA to design signal locations. GSV uses the signals in this file.

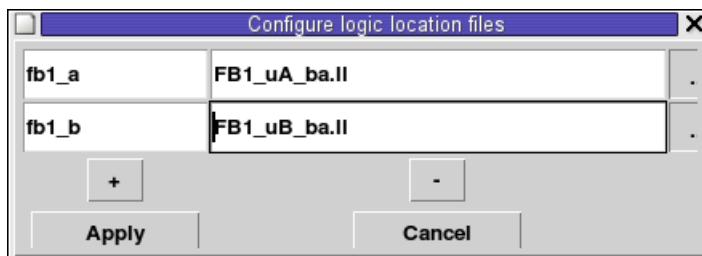
3. Import the results from Vivado and backannotate results from place and route:

- Return to the map database state in the prototyping tool and run the database apply_state -import_vivado command. This command imports the Vivado .ll file and place-and-route results into the prototyping database.
- Run the database apply_state -backannotate command with the -gsv option to map the objects in the Vivado .ll file to the prototyping mapped database. GSV uses the Vivado .ll file to map values back to the RTL.

```
database apply_state -backannotate gsv
```

4. Set up files for debug, including information from the .ll file:

- Run the export runtime command to generate a GSV database that includes the Vivado and backannotated .ll files under `exportResultsDir/system/readback`. The command also creates a parallel debug directory with the debug database in the same system export directory.
- Start the ProtoCompiler runtime executable (`protocompiler_runtime debug`).
- Open the GSV project in the exported `readback` folder, and apply the backannotated .ll files using the Readback IICE dialog box. (Readback icon (RR) in the GSV IICE). FPGA names must follow the naming convention for individual FPGAs: `fb1_a, fb1_b....`



- When used in multi-design mode (MDM), the FPGA names must be preceded by the device number. For example: `d9_fb1_a, fb1_b....`

5. Run debug.

- Arm the readback IICE to perform a single readback operation. Signal values are overlaid in the RTL view per signal line.
- Run debug. If you instrumented the design, the command runs debug and GSV together.

- To append samples to the waveform for asynchronous readback, use the readback sample_buffer -append command.

For information about the syntax for the readback and run commands, see *Instrumentor and Debugger Commands*, on page 15 in the *HAPS Proto-Compiler Debugging Environment Reference*.

Limitation

Does not support VM flow for backannotation.

Use Model Example: Synchronous GSV for a Single-FPGA Design

This following procedure describes a use model which uses synchronous GSV for a single, unpartitioned FPGA, with a DTD IICE for triggering. For multi-FPGA partitions, follow the steps described in *Use Model Example: Synchronous GSV with a Multi-FPGA Design*, on page 762.

1. Set up the hardware as for DTD2, using a HAPS-DX system as a debug hub, and with the requisite MGB riser and SATA daughter boards, SATA crossover cables, coax cables, and HT3 con cable.

See *Setting up the Hardware for GSV*, on page 772.

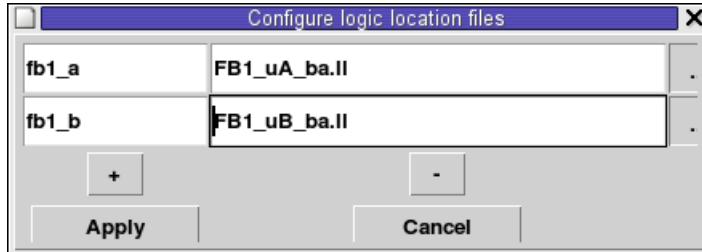
2. Set up the design for register GSV.
 - Generate a root database for the design.
 - Specify option set prepare_readback 1 to enable the GSV flow. To enable the eGSV flow, the prepare_readback option must be set to 2.
3. Continue with the normal design flow through the pre-instrument, compile and map stages.

To run debug with GSV, you must run the pre-instrument phase, even if you do not actually instrument any signals. The debug directory, which is required for debug, is generated at this phase.

4. Place and route as usual, using the launch vivado and export vivado commands.

The place-and-route tool generates a logic location (.ll) file, based on the GSV information. This file maps the register information from the FPGA to design signal locations. It contains all matched signals, and all signals in this file are automatically used for GSV.

5. Set up files for debug, including information from the .ll file:
 - Return to the map database state in the prototyping tool and run the database apply_state -import_vivado command. This command imports the Vivado .ll file and place-and-route results into the prototyping database.
 - Run the database apply_state -backannotate command to map the objects in the Vivado .ll file to the prototyping mapped database. This back-annotated file is called *_ba.ll.
6. Run debug.
 - Run the export runtime command to generate a GSV database that includes the Vivado and backannotated .ll files under `exportResultsDir/system/readback`. It also creates a parallel debug directory with the debug database in the same system export directory.
 - Open the GSV project in the exported `readback` folder, and apply the backannotated ll files using the Readback IICE dialog box. (Readback icon (RR) in the GSV IICE). FPGA names must follow the naming convention for individual FPGAs: `fb1_a`, `fb1_b`....



- When used in multi-design mode (MDM), prefix the FPGA names with the device number. For example: `d9_fb1_a`, `fb1_b`....
- Start the ProtoCompiler runtime executable, and run debug (`protocompiler_runtime debug`). If you instrumented the design, the command runs debug and GSV together.

The debugger reads back the register values of the matched signals and applies them in the source code viewer in the debugger. It also lists unmatched ll signals under LL tree in same view. After GSV is complete, the values are appended to the signal names.

Use Model Example: Synchronous GSV with a Multi-FPGA Design

This following procedure describes a use model which uses synchronous GSV for a multi-FPGA design, with a DTD IICE for triggering. For single-FPGA partitions, follow the steps described in [Use Model Example: Synchronous GSV for a Single-FPGA Design, on page 760](#).

1. Set up the hardware as for DTD2.

See [Setting up the Hardware for GSV, on page 772](#).

2. Prepare the design for GSV.

- To use the clock control capability, instantiate a CCM, and use the generated clocks from the CCM. (See [Instantiating and Configuring Clock Control Modules for GSV, on page 764](#) for details.) If you do not instantiate the CCM, there is no guarantee that the state information is aligned with the same system clock cycle.
- Generate a root database.
- Specify the option set `prepare_readback 1` command to enable GSV.

3. Continue with the normal design flow through the pre-instrument and compile stages.

You must run pre-instrument. To run GSV with debug and clock control, you must instrument the signals to enable DTD2, and configure the DTD IICE with the export trigger option enabled. The export trigger signal serves as an assertion to the CCM to signal the start of a readback operation.

- Configure the instrumentor for the DTD2 flow.
- Set the export trigger option:

```
iice controller -iice {IICE_0} -exporttrigger 1
```

- Compile the design.

4. Follow these steps to set up the hardware and partition the design.

- For the hardware, set it up for deep trace debug with DTD2. DTD2 uses a HAPS-70 system with a HAPS-DX system as a debug hub. Also set up the requisite MGB riser and SATA daughter boards, SATA crossover cables, coax cables, and HT3 con cable. See [Setting up the Hardware for GSV, on page 772](#) for a description of the setup, and

[Using a Debug Hub with SATA \(DTD2\), on page 716](#) for information about deep trace debug.

- If you are using a CCM, do the following additional steps at the pre-partition and partition stages:

| | |
|---------------|--|
| Pre-partition | <ul style="list-style-type: none">• In the tss file, assign the trigger input port to the trace used to gate the CCM. You must connect the J1 connector on the HAPS DX7 system to an eligible HT3 connector on the HAPS-70 system with an HT3 cable, so that the clock enable signals can be distributed across the whole FPGA. See Setting up the Hardware for GSV , on page 772 for details. |
| Partition | <ul style="list-style-type: none">• Replicate the clock divider modules from the CCM to ensure that signals are synchronized by adding assignment and replication constraints to the pcf file.• Assign the trigger input port, to gate the CCM. For the hardware, use a HT3 cable to connect the HAPS-DX7 J1 connector to the A{n} connector on FPGA A on the HAPS-70 system.• Assign the clock enable to GCLK network. <p>See Instantiating and Configuring Clock Control Modules for GSV , on page 764 and Running GSV with Clock Control , on page 773 for details.</p> |

- Run system route as usual.
5. Run system generate and do the following for each FPGA generated:
- Launch the ProtoCompiler tool for the individual FPGA.
 - Synthesize, place, and route the individual FPGA as usual. The GSV capability uses the same pins (GPIO and HT3), that are usually configured as user design pins. When GSV is enabled, the tool ensures that there is no configuration conflict.
- When it generates the bit file, the place-and-route tool generates the logic location (.ll) file, which maps the register information from the FPGA to design signal locations. The .ll file contains all matched signals, and all signals in this file are automatically used for GSV.
6. Use the `export runtime` command to export the bit files of the individual FPGA, the two debug and GSV projects, and backannotated ll files back to the `export_results` directory.

The tool creates two directories, one for debug and one for GSV, with the appropriate files.

7. Return to the System Generate state and link the partitioned FPGA to a corresponding mapped state of a single FPGA, using the database `apply_state -link_module` command.
8. Map Vivado .ll files to RTL-compatible .ll files (_ba)

```
database apply_state -backannotate
```

This command backannotates information from the .ll files generated after place and route back to the map stage. Use these backannotated *_ba.ll files for GSV.

9. Run debug.

- From the System Generate state, run `export runtime` to export all debug files, including the .ll files for the FPGAs. The tool exports a single SoC project instead of multiple files for the FPGAs.

There are two databases created, with the files located in the debug and readback directories under `export_results/directory`. The readback directory is generated if `prepare_readback` was set, and contains the files for GSV debug, along with all .ll files. The debug directory contains the instrumented database files needed to run deep trace debug.

- Start debug using the Protocompiler runtime executable:

```
protocompiler_runtime debug
```

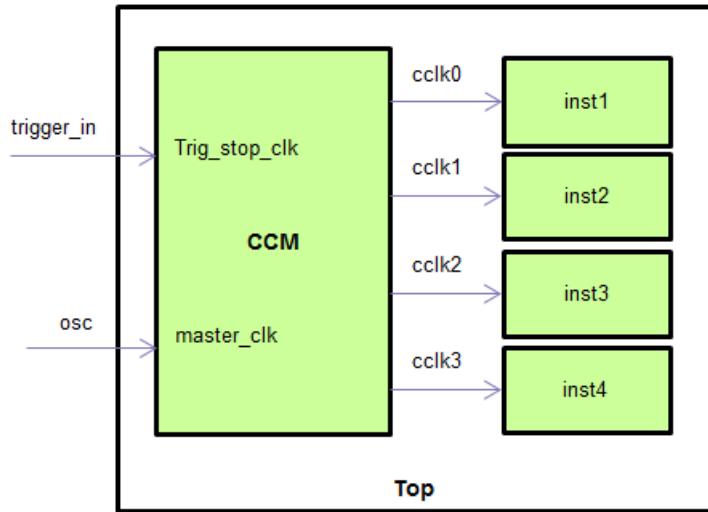
For details about this process, see [Running GSV with Clock Control, on page 773](#).

The debugger reads back the register values of the matched signals and applies them in the source code viewer. After GSV is complete, the tool appends the values to the signal names.

Instantiating and Configuring Clock Control Modules for GSV

To ensure that data can be read synchronously across the FPGA and that register GSV is reliable, you must stop the design clock. The clock control module (CCM) that governs this process uses an uncontrolled input clock as the master clock and derives generated clocks from it to drive the instances in a synchronized fashion. The inputs to the CCM are a master clock and a

clock stop, and the outputs are four clocks generated from a single master source, as shown in the figure below. One example of a CCM is MMCM (mixed-mode clock manager).



Do the following to use the CCM:

1. Instantiate the CCM in your top-level design.
 - Get the CCM source file:
protocompiler_installDir/lib/di/haps_controlled_clocks.v
 - Edit the CCM parameters that control the phase and frequency of uncontrolled clocks to values for your design. Do not edit any other CCM parameters. The following table illustrates the parameters to be edited.

| Parameter | Description |
|-----------------------------|---|
| CCLK{0 to 3}_COUNTER_MAX | <p>Specifies an integer value to control the period of the generated clocks. The following equation where M is the parameter value, specifies its relation to the master clock:</p> $T_{CCLK}(n) = 2*(M + 1)*T_{master_clk}$ <p>If the master clock period is 10 ns and the parameter value M is 3, then the generated clock period is 8*10 ns.</p> <p>Default = 2</p> |
| CCLK{0 to 3} _COUNTER_PHASE | <p>Specifies a value to control the phase of the generated clock. Define the phase of each generated clock with respect to the smallest phase value among the four clocks.</p> <p>Default = 0</p> |
| UMR_CLK_CTRL_CAPIM_ADDRESS | <p>Sets the CAPIM address for the CAPIM that is used within CCM.</p> <p>Default =30</p> |

- Instantiate the CCM in the top-level design.
2. For multi-FPGA designs, add the following constraints to the pcf file to replicate the CCM across all FPGAs driven by clkout [3:0]:
- Assign the clock module to FPGA A with assign_cell PCF commands:
- ```
assign_cell clock_control systemName.uA
assign_cell clock_control.te systemName.uA
assign_cell clock_control.haps_clk_ctrl inst systemName.uA
```
- Use replicate\_cell commands to replicate just the clock generate modules in all the FPGAs, including FPGA A. The commands below replicate clock generator modules clkgen0, clkgen1, clkgen2, and clkgen3 in FPGAs A and B.

```

replicate_cell clock_control.cclk_gen_to_be_replicated.cclkgen0
{FB1.uA FB1.uB}
replicate_cell clock_control.cclk_gen_to_be_replicated.cclkgen1
{FB1.uA FB1.uB}
replicate_cell clock_control.cclk_gen_to_be_replicated.cclkgen2
{FB1.uA FB1.uB}
replicate_cell clock_control.cclk_gen_to_be_replicated.cclkgen3
{FB1.uA FB1.uB}

```

- Assign the clock control enable signal to the GCLK network to ensure that the signal is uniformly distributed to all the replicated clock generator modules across the FPGAs. The following example assigns the clock enable to GCLK4, whose source is in FPGA A:

```

assign_cell {clock_control.clk_cntr_en_keeper}
{FB1.clk.bd_clk4}
net_attribute {clock_control.clk_cntr_en} -function GCLK
-diffsingle -is_clock 1
net_attribute {clock_control.clk_cntr_en_int} -function GCLK

```

- Use the **assign\_port** command to specify the hardware connection to export the external trigger signal which stops the clock. Make sure it feeds the FPGA that contains the CCM.

In the following example, the clock is exported from the external HAPS-70 DX system through a HT3 cable. On the HAPS-DX\_S4 system, the board specification locks the exported trigger signal to the BD14 pin on J6. On the HAPS-70 side, it can be connected to the seventh pin of the A blade of any suitable HT3 connector; in this case it is connected to the seventh pin of the A5 connector:

```
assign_port {trigger_in} -trace {FB1_A5_A[7]}
```

3. For multi-FPGA designs, define the GCLK to be used as FPGA source in the tss file:

```
board_system_configure -clock {FB1.GCLK4} FPGA
```

This command distributes the clock enable signals over the GCLK network.

## Memory GSV

This feature supports readback of an RTL memory array using the Verilog or VHDL hierarchy path in the ProtoCompiler runtime. To perform readback, the user defined memory must be mapped to the Xilinx BRAM.

To perform the readback of the memory, execute the following command in the ProtoCompiler runtime:

```
readback memory -mem {memoryName} [-file fileName] [-format bin|hex]
```

For more information on the command and its syntax, see readback command in HAPS Prototyping Debugging Environment Reference User Guide.

## Example: Implementing Memory GSV for a Single-FPGA Design

This following procedure describes a use model which uses asynchronous memory GSV for a single unpartitioned FPGA.

1. Define the memory in the RTL file.

```
reg [data_width - 1 : 0] mem_tdpram [(2**addr_width) - 1 : 0];
```

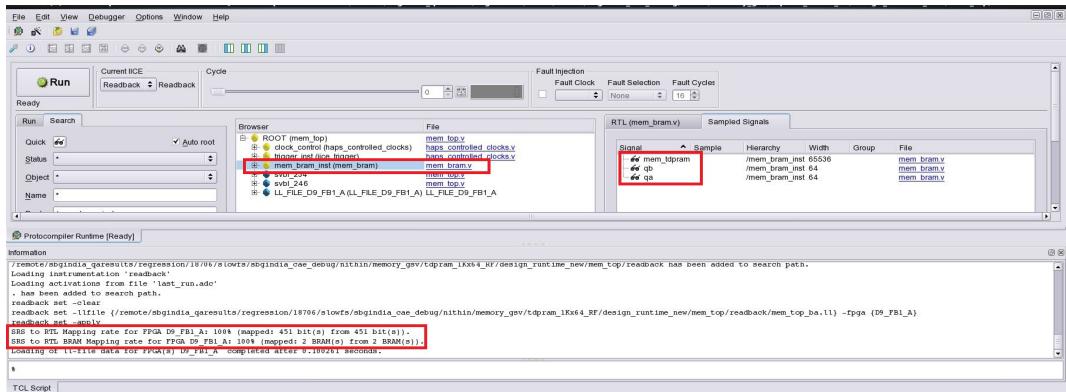
2. Provide the following attribute in the .fdc file to stop memory optimization:

To stop different ROM and RAM optimizations, set `syn_preserve = 1` on the user-defined memory instance in the FDC file as shown in the example below. For memories defined in a hierarchy, the attribute must be specified for the complete hierarchical instance.

```
define_attribute {i:mem_tdpram[63:0]} {syn_preserve} {1}
```

3. Use the regular GSV flow to generate the \*ba.ll file containing the mapped signals and memory information.
4. To perform readback of a memory, configure the device, setup the clocks, and load the readback project from `exportResultsDir/system/readback/debug.prj` in the ProtoCompiler Runtime.
5. Click the  icon and load the \*\_ba.ll file.

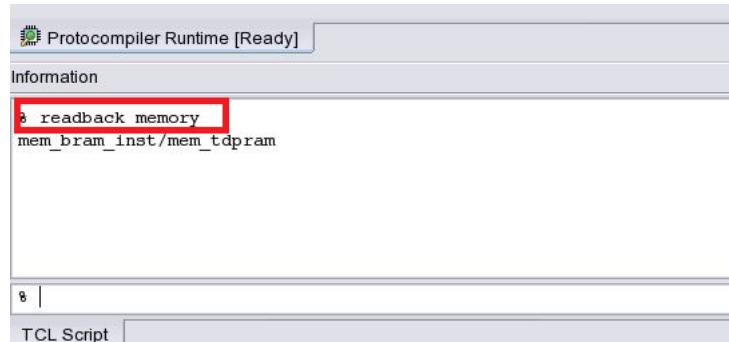
When the \*\_ba.ll file is loaded, the number of mapped memories and the number of registers in the design are displayed in the console window. The mapped RTL memory and signals with the hierarchy information are displayed in the Sampled Signals tab.



## 6. Use the readback command.

- You can use the existing readback command to stop or release CCM clocks.
- To list the different RTL memories in the design along with their hierarchical paths in the Tcl Console window, execute the following command:

```
readback memory
```



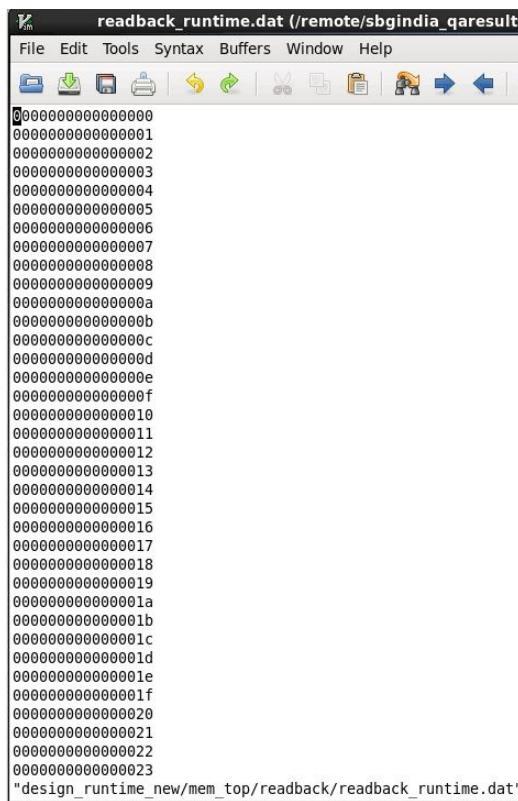
## 7. Use the following commands to read back the contents of a specific memory:

- To readback the memory and view the data in Tcl Console window:

```
readback memory -mem /mem bram inst/mem tdpram
```

- To read back the memory and view the hex formatted data in a file:

```
readback memory -mem /mem_bram_inst/mem_tdprom -file
runtime data.dat -format hex
```



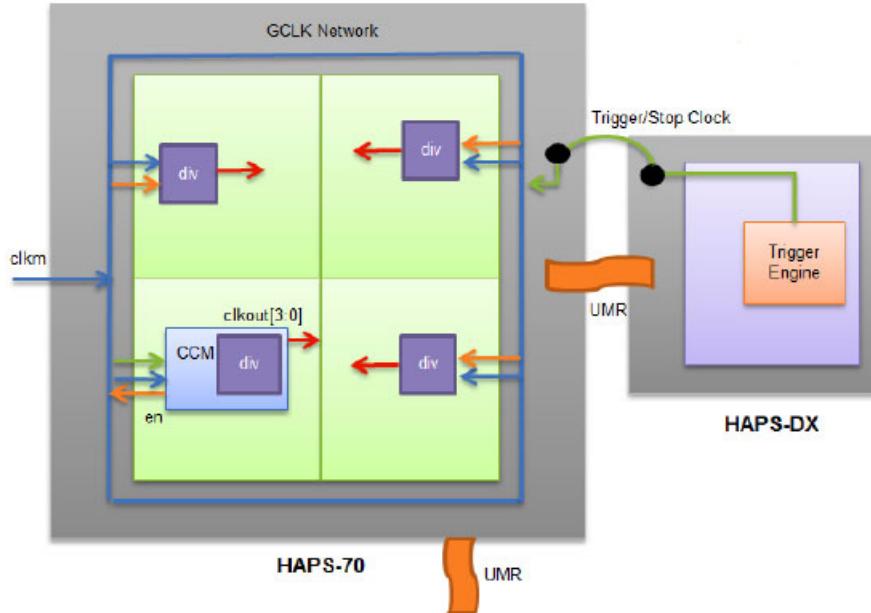
- To read back the memory and view the bin formatted data in a file:

```
readback memory -mem /mem_bram_inst/mem_tdprom -file
runtime data.dat -format bin
```

- By default, the readback file is saved in the readback folder. To save the file to another location, provide the path. For example:  
`readback memory -mem /mem_bram_inst/mem_tdpram -file  
<path>/runtime data.dat -format hex`

## Setting up the Hardware for GSV

The required hardware for GSV is a HAPS-70 and a HAPS-DX system, set up as for DTD2. The following figure shows the connections:



Connect the hardware:

1. Connect the master clock to any global clock in the pcf file. For example: GCLK2.
2. Use a con cable to connect the HAPS-DX system to the HAPS-70 system to export the trigger/stop\_clock.

For example, connect the J6 connector of the HAPS-DX system to the A5 HT3 connector of the HAPS-70 FPGA A module with the con50 cable.

3. Set up for DTD2 multi-FPGA deep trace debug:
  - Use the coax cable to connect the HAPS-70 PLL out1 to the PLL IN input on the HAPS-DX7 system.
  - Use any global clock except the master clock (for example, GCLK3) as the reference clock for deep trace debug. Specify this during instrumentation.

4. Build the setup for deep trace debug according to the instrumentor log report, which is generated after the system generate state.

This is an example of the report:

```
Multi-FPGA DTD Module summary for IICE_IICE_0:

```

```
Instrumentation Memory Connections: default (HAPS-DX7_S4)
```

```
Links Total: 2
```

```
MGB card: SATA-4_HS_MGB
```

```
2 Connections Total:
```

```
 MGB1.J2.X5 - Must be connected
```

```
1 of the following 7 must be connected
```

```
 MGB1.J2.X2
```

```
 MGB1.J2.X3
```

```
 MGB1.J2.X4
```

```
 MGB1.J3.X2
```

```
 MGB1.J3.X3
```

```
 MGB1.J3.X4
```

```
 MGB1.J3.X5
```

```
User Design Connections: HAPS-70
```

```
2 Required Connections:
```

```
 FB1.uA:
```

```
 MGB1.J2.X5
```

```
 FB1.uB:
```

```
 MGB1.J2.X5
```

```
MGB connection mux ratio: 2
```

```
Maximum sample clock frequency: 35.000000 MHz
```

5. Configure all the HAPS systems with correct voltage, GCLK frequency settings and FPGA configurations in Confpro.

Connect the PC to the HAPS system. In the example the UMRBus is used to connect the systems.

## Running GSV with Clock Control

Follow this sequence of steps to run GSV with clock control:

1. Launch a debugger session and load the debug project generated by export runtime.

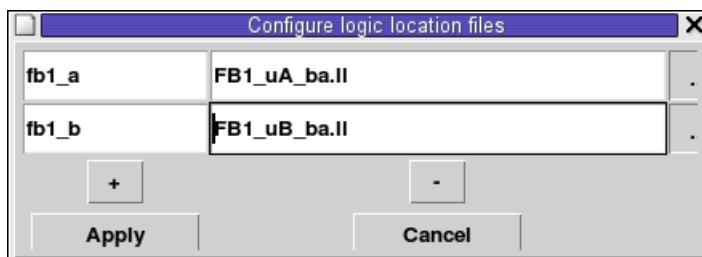
2. Train the GTX links for deep trace debug setup, either from the GUI or the Tcl prompt.
    - From the GUI, open the project for the debug IICE, and click the IICE icon. In the Enhanced settings for IICE unit dialog box, click Check and setup Multi-FPGA HDTD Links.
    - Alternatively, type this command at the Tcl prompt:

haps setup mfhdtd links

If the GTX links are correctly trained, the following message is reported. If you do not see this message, there could be problem with the setup.

3. Launch the second debugger session and load the readback project in this exported directory: `export_results/system/readback`
  4. In the GSV IICE, click the Readback icon () to load the backannotated logic location files for the individual FPGAs, as shown below.

Use this FPGA naming convention: *boardName\_fpgaName*. The *boardName* must be fb1, fb2, fb3 ..., and the *fpgaName* must be A/a, B/b, C/c and D/d.



5. In the GSV IICE, type the following run command with the -readback argument to start the operation.

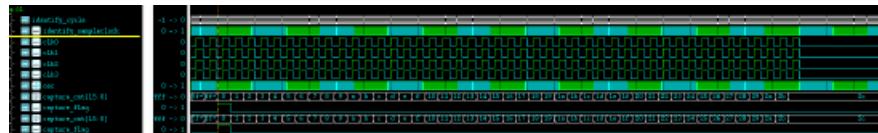
For example:

run -iice readback -readback 2 -stepping 2 -wait for trigger 1 -release clock 1

This command performs three readbacks, and lets the controlled clocks tick two times between successive readbacks. The readbacks wait for the trigger and all the controlled clocks are released after GSV is completed. Alternatively you can use the readback command instead of run. For the complete command syntax, see [Instrumentor and Debugger Commands, on page 15](#) in the *Debugger Environment Reference*.

The GSV IICE waits for the HAPS-DX system to assert the export trigger. When the debug IICE gets a trigger, it exports a trigger pulse from the HAPS-DX system to the HAPS-70 system.

6. Switch to debugger session where the debug project is loaded and do the following to arm the trigger.
  - Arm the trigger condition.
  - Click Run. When the debug IICE is triggered, the GSV IICE gets triggered and the GSV sample is downloaded in the GSV IICE. The GSV IICE receives the trigger signal some clock cycles of the master clock after the point where the trigger has occurred. The debug waveform below shows the point where the export trigger is received and the controlled clocks have stopped to perform GSV, using clock stepping mode.



The next GSV waveform shows the point where the GSV is performed by stepping the clock



7. Write out the GSV waveform by running this command in the GSV IICE:

```
write fsdb file_name.fsdb
```

You must use this command to dump out the GSV waveform for viewing GSV data because the waveform viewer icon is not available at this point.

# GSV or eGSV Flow Using ZCEI based CCM on HAPS-100

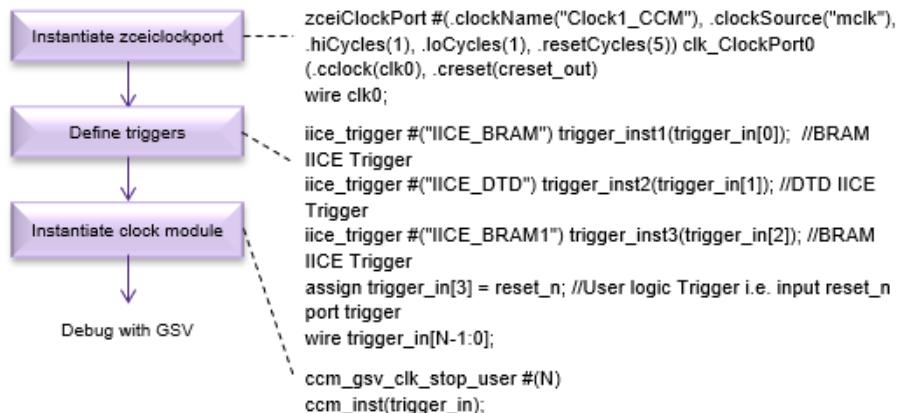
The following flow for HAPS-100 designs describes how to use zcei clock ports to stop the clocks and enable readback capture for GSV or eGSV. The following procedure describes an example that uses zcei clock ports to run synchronized GSV or eGSV debug on a two-FPGA design.

See these sections for details about the flow:

- [Setting up to Run GSV or eGSV with HAPS-100, on page 776](#)
- [Debugging a HAPS-100 Design Using GSV or eGSV, on page 780](#)
- [Limitations, on page 783](#)
- [Examples, on page 784](#)
- [Implementation and Execution, on page 794](#)

## Setting up to Run GSV or eGSV with HAPS-100

This figure summarizes the main setup steps. Refer to the subsequent procedure for detailed steps.



1. Enable readback by setting the `prepare_readback` option:
  - For GSV, set this value: `set option prepare_readback 1`

- For eGSV, set this value: set option prepare\_readback 2

eGSV achieves faster readback download speeds than GSV.

## 2. Set up the RTL files:

- Explicitly include these .v files for the clock port along with the other source files.

```
<build_path>/lib/synip/clocks/gsv_trigger_hypermods.v
<build_path>/lib/synip/hcei/zceistubs.v
```

- Instantiate zcei clock ports in the RTL. The number of clock ports depends on how many clocks need to be generated for the design. For example, if you want to generate four clocks (i.e. divide by 2, 4, 8 and 16), add four zceiClockPort definitions, as shown in the following example.

```
wire clk1;
wire clk2;
wire clk3;
wire clk4;

zceiClockPort #(.clockName("Clock1_CCM"), .clockSource("mclk"),
.hiCycles(1), .loCycles(1), .resetCycles(5)) clk_ClockPort0
(.cclock(clk1), .creset(creset_out));

zceiClockPort #(.clockName("Clock2_CCM"), .clockSource("mclk"),
.hiCycles(2), .loCycles(2), .resetCycles(5)) clk_ClockPort1
(.cclock(clk2));

zceiClockPort #(.clockName("Clock3_CCM"), .clockSource("mclk"),
.hiCycles(4), .loCycles(4), .resetCycles(5)) clk_ClockPort2
(.cclock(clk3));

zceiClockPort #(.clockName("Clock4_CCM"), .clockSource("mclk"),
.hiCycles(8), .loCycles(8), .resetCycles(5)) clk_ClockPort3
(.cclock(clk4));
```

Here, the base clock, i.e. the clock input source to the zcei clock ports, is defined as mclk. Use the clock name from the constraint instead of the RTL clock port name when defining clocks that must be controlled or stopped for GSV capture. For example, if the top-level RTL clock port input is osc, and the FDC file defines the clock constraint as mclk (see below), use the corresponding clock constraint name from the FDC file (mclk) to define the clock source to the zceiclockport, not osc.

```
create_clock -name {mclk} {p:osc} -period {50}
```

### 3. Define the triggers.

- Define up to a maximum of 10 user triggers in the idc file. You can specify a combination of triggers from different IICE blocks, and user logic triggers.

IICE Triggers:

```
iice_trigger #("IICE_0") trigger_inst1(trigger_iice_0);
iice_trigger #("IICE_1") trigger_inst2(trigger_iice_1);
iice_trigger #("IICE_2") trigger_inst3(trigger_iice_2);
iice_trigger #("IICE_3") trigger_inst4(trigger_iice_3);
iice_trigger #("IICE_4") trigger_inst5(trigger_iice_4);
iice_trigger #("IICE_5") trigger_inst6(trigger_iice_5);
iice_trigger #("IICE_6") trigger_inst7(trigger_iice_6)
```

User logic triggers:

```
assign trigger_in[0] = trigger_iice_0;
assign trigger_in[1] = trigger_iice_1;
assign trigger_in[2] = trigger_iice_2;
assign trigger_in[3] = trigger_iice_3;
assign trigger_in[4] = trigger_iice_4;
assign trigger_in[5] = trigger_iice_5;
assign trigger_in[6] = trigger_iice_6;
assign trigger_in[7] = gpio_1; // User logic trigger
assign trigger_in[8] = gpio_4; // user logic trigger
assign trigger_in[9] = gpio_7 // user logic trigger
```

- Connect the triggers to a wire called `trigger_in[9:0]`, as shown above in the example.
- Next, connect the triggers to a user GSV clock stop trigger module as shown below.

```
ccm_gsv_clk_stop_user #(10) clk_stop_inst (trigger_in);
```

The number 10 defines the trigger input parameters, and the current maximum is 10 triggers.

The control clock module (CCM) is responsible for stopping all the CCM clocks to enable GSV capture. At runtime, use the clock module to select which of the 10 triggers to use to stop the clocks. You can also use it to combine different triggers by masking them with a MCAPIM register. By default, all triggers are ORed.

### 4. Define the CCM base clock out in the TSS file.

The CCM base clock out is routed to other FPGAs as the FPGA clock source.

- Define the clock routing to other FPGAs. This example shows the TSS definition for the master clock source, osc. The master clock source uses GCLK1 in the PLL1 as the clock source, and is routed to both FPGAs A and B in this 2-FPGA design:

```
board_system_configure -pll mb1.PLL1.CLK1 -frequency 20000
-name d_clk
board_system_configure -clock {mb1.uA.CLK1} d_clk
board_system_configure -clock {mb1.uB.CLK1} d_clk
```

- Define the FPGA clock source for the CCM clock base out. The following example defines CLK3 as the FPGA clock source.

```
board_system_configure -clk_src mb1.uA.CLK_SRC3 -name ccm_clk1
-frequency 10000
board_system_configure -clock {mb1.uA.CLK3} ccm_clk1
board_system_configure -clock {mb1.uB.CLK3} ccm_clk1
```

## 5. Add constraints in the PCF file for the clock module and clocks:

- Define the FPGA where the clock module will be instantiated. This PCF constraint specifies that it be instantiated in FPGA A:

```
zcei_assign_route -master_bin mb1.uA
```

- Define the PCF constraints for the master clock and the FPGA clock source as shown in the PCF commands below.

Definition for CCM clock input:

```
net_attribute osc -function GCLK
assign_global_net osc d_clk
```

Definition of CCM clock base out as FPGA clock source:

```
net_attribute {haps_infra_clksrc_mclk_cclock_base_out}
-function GCLK
assign_global_net {haps_infra_clksrc_mclk_cclock_base_out}
ccm_clk1
```

## 6. Run through the regular GSV flow through place and route, to generate the `.ll` file and bit files.

You can now use GSV to debug a HAPS-100 design.

## Debugging a HAPS-100 Design Using GSV or eGSV

1. Set up the options for the flow and set up the triggers as described in [Setting up to Run GSV or eGSV with HAPS-100, on page 776](#).
2. Run through the regular GSV flow through place and route, to generate the .ll file and bit files.
3. To perform readback of a memory, do the following:
  - Configure the device.
  - Set up the clocks.
  - Start the debugger and load the readback project from `exportResultsDir/system/readback/debug.prj`.
4. Load the \*.ll file, or \*.ba.ll file if the design is backannotated.
  - Click the RR icon.
  - Load the files using their FPGA names:

`fb1.uA` → To load `fb1_uA.ll/fb1_uA_ba.ll`  
`fb1.uB` → To load `fb2_uB.ll/fb1_uB_ba.ll`

Note that you do not have to provide the device number, just the FPGA name. When you load the file, the tool displays the number of mapped memories and registers in the design in the console window.

5. Use the options in the GUI to stop the clocks and perform readback, as normal.
6. To mask bits of a trigger source during runtime, issue this command in the runtime shell to enable or disable individual bits:

```
ccm trigger -any -mask <32-bit_hex_value>
```

You can only write to bits [9:0] of the mask value. Each of the writable bits can be individually masked (1) or unmasked (0). The default is 0. For example: To mask the trigger source bits 0, 4, and 5, use this command:

```
ccm trigger -any -mask 110001
```

You can also use the GUI to enter this information:



The CCM block contains a register which can be accessed through Confpro. This register has two functions. The MSB bit defines the operation to run on the triggers: OR (0) or AND (1). Mask bits enable/disable specific trigger inputs. The default is 0, and defines how many triggers are active. If a trigger is to be disabled, mask it by setting it to 1.

7. To synchronize and start the clocks at the same time on all the FPGAs, initialize the CCM by following these steps.
  - Create an HMF file that contains the serial number of the HAPS system. The following is an example of a `haps_system.hmf` file, with serial number X007837.

```
{
 "tsdmaphaps": {
 "FB1": {"serial": "X007837"}
 }
}
```

- Start the Confpro shell:

```
<build>/bin/confprosh
```

- Run the `proto_rt` command to initialize the CCM.

```
proto_rt::run_ipinfra -hmf hmf_file -train all
```

- Generate a report:

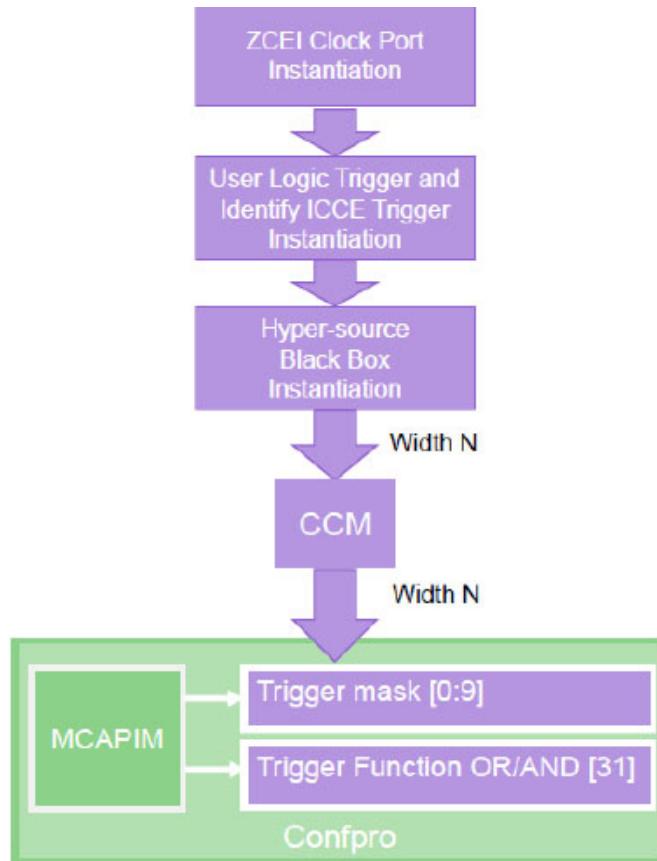
```
proto_rt::run_ipinfra -hmf hmf_file -report all -file file.txt
```

- Check for the following snippet in the CCM report, to confirm that the CCM is initialized.

```
===== CCM Report =====
CCM "/umr3usb0/mod-X007837/uf1/usr/DBG_CCM" is Initialized
=====
```

## Summary of GSV Implementation

The following figure summarizes the GSV implementation. The CCM block contains a register that you can access with the Confpro configuration software. The register has two functions: trigger function and trigger mask. The MSB determines whether the triggers are ORed (0) or ANDed (1). The remaining bits are mask bits that enable or disable specific trigger inputs. The default is 0, which defines how many triggers are active. To disable a trigger, set it to 1.



The last step in the procedure mentions some other capim, not mcapim. It's a bit confusing when the figure doesn't match the text.

## Limitations

- For multiple trigger inputs to the clock module, only the OR operation is supported.
- For clock generation with `zceiClockPort`, you can only have a single base clock. This means that the `clockSource()` value must be the same for all clocks.
- Phase shifting of generated clocks relative to the base clock is not supported.

- With a mixed-system hardware setup, you cannot insert new ZCEI clock module on HAPS-80 systems. The CCM must be on a HAPS-100 system.
- Combination of options `prepare_readback 1` and `prepare_readback 2` is not supported.
- Input value for clock divider is supported only in integer format. Fractional and decimal formats are not supported.

## Examples

### `zceiClockPort` Example (Four Clocks)

This example uses using four `zceiClockPort` definitions in the RTL to generate four clocks (divide by 2, 4, 8 and 16):

```
wire clk1;
wire clk2;
wire clk3;
wire clk4;

zceiClockPort #(.clockName("Clock1_CCM"), .clockSource("mclk"),
 .hiCycles(1), .loCycles(1), .resetCycles(5)) clk_ClockPort0
 (.cclock(clk1), .creset(creset_out));

zceiClockPort #(.clockName("Clock2_CCM"), .clockSource("mclk"),
 .hiCycles(2), .loCycles(2), .resetCycles(5)) clk_ClockPort1
 (.cclock(clk2));

zceiClockPort #(.clockName("Clock3_CCM"), .clockSource("mclk"),
 .hiCycles(4), .loCycles(4), .resetCycles(5)) clk_ClockPort2
 (.cclock(clk3));

zceiClockPort #(.clockName("Clock4_CCM"), .clockSource("mclk"),
 .hiCycles(8), .loCycles(8), .resetCycles(5)) clk_ClockPort3
 (.cclock(clk4));
```

Note that `mclk` is the clock input source to the zCEI clock ports and is called the base clock. You must provide the clock constraint name instead of the clock port name as the clock source. For example, if the top level clock port input is `osc`, you use the clock constraint name provided in the fdc as the clock source to the `zceiClockPort` command.

In the following example `mclk` refers to the clock constraint name for the clock port `osc`. This clock port should be used as the `clockSource` name in the zCEI clock port definition.

```
create_clock -name {mclk} {p:osc} -period {50}
```

## IICE Trigger Setup Example

This example defines the user logic and IICE triggers, and connects them to a wire called trigger\_in[9:0] as shown in the example.

```
iice_trigger #("IICE_0") trigger_inst1(trigger_iice_0);
iice_trigger #("IICE_1") trigger_inst2(trigger_iice_1);
iice_trigger #("IICE_2") trigger_inst3(trigger_iice_2);
iice_trigger #("IICE_3") trigger_inst4(trigger_iice_3);
iice_trigger #("IICE_4") trigger_inst5(trigger_iice_4);
iice_trigger #("IICE_5") trigger_inst6(trigger_iice_5);
iice_trigger #("IICE_6") trigger_inst7(trigger_iice_6)

#User logic triggers
assign trigger_in[0] = trigger_iice_0;
assign trigger_in[1] = trigger_iice_1;
assign trigger_in[2] = trigger_iice_2;
assign trigger_in[3] = trigger_iice_3;
assign trigger_in[4] = trigger_iice_4;
assign trigger_in[5] = trigger_iice_5;
assign trigger_in[6] = trigger_iice_6;

assign trigger_in[7] = gpio_1;
assign trigger_in[8] = gpio_4;
assign trigger_in[9] = gpio_7
```

## GSV/eGSV Flow Using HAPS-100 Clock Generator

(*HAPS-100 Only*)

This section describes the GSV and eGSV flow to stop the clocks and enable the readback capture in HAPS-100 devices using the HAPS Clockgen. The following procedure describes a use model for GSV/eGSV flow for a two-FPGA Synchronous GSV flow:

1. Enable GSV/eGSV flow, HAPS clock generator, and other required options:
  - Enable readback by setting the prepare\_readback option:
    - For GSV, set this value: set option prepare\_readback 1
    - For eGSV, set this value: set option prepare\_readback 2

eGSV achieves faster readback download speeds than GSV.

- Enable RTL instrumentation through \$dumpvars.

```
option set debug_dumpvars 1
```

- Enable the HAPS clock generator mode.

```
option set haps_clockgen_mode 1
```

- Enable Verdi correlation. (Mandatory for UC2 flow).

```
option set verdi_mode 1
```

2. Set up the RTL files to insert the new clock generator in the design and set up the CCM Clock:

- For Unified Compile flow, explicitly include the following .v files for the clock port along with other source files:

```
<build_path>/lib/synip/clocks/gsv_trigger_hypermods.v
<build_path>/lib/synip/hcei/zceistubs.v
```

- Declare the CCM source using the TSS command. For example:

```
board_system_configure -create_ccm_clocks -source mb1.uA -load
{mb1.uA mb1.uB}
```

This ensures that the CCM source resides in FPGAA and the CCM load is replicated on both FPGAA and FPGA B.

- Create CCM Clocks. To create CCM clocks, define the clock constraint in the FDC file with the required frequency and add it to the design as input by declaring the clock port input. For example:

```
create_clock -name Clock1_CCM [get_ports {p:clk0}] -period 40 -waveform
{0 20}
create_clockgen_group {CLOCK1_CCM} -group {grp1}
```

In this example, a frequency of 25 MHz is generated and is fed as input to the design through top level clock port clk0.

- To connect the triggers to the clock stop input of the CCM, manually instantiate the triggers and CCM clock stop modules in the RTL as shown in the example. You can define up to a maximum of 10 user triggers in the idc file. These triggers can be a combination of triggers from different IICE blocks, or user logic triggers.

Example for IICE Triggers:

```
iice_trigger #("IICE_0") trigger_inst1(trigger_iice_0);
iice_trigger #("IICE_1") trigger_inst2(trigger_iice_1);
iice_trigger #("IICE_2") trigger_inst3(trigger_iice_2);
iice_trigger #("IICE_3") trigger_inst4(trigger_iice_3);
iice_trigger #("IICE_4") trigger_inst5(trigger_iice_4);
iice_trigger #("IICE_5") trigger_inst6(trigger_iice_5);
iice_trigger #("IICE_6") trigger_inst7(trigger_iice_6)
```

Example for User Logic Triggers:

```
assign trigger_in[0] = trigger_iice_0;
assign trigger_in[1] = trigger_iice_1;
assign trigger_in[2] = trigger_iice_2;
assign trigger_in[3] = trigger_iice_3;
assign trigger_in[4] = trigger_iice_4;
assign trigger_in[5] = trigger_iice_5;
assign trigger_in[6] = trigger_iice_6;
assign trigger_in[7] = gpio_1; // User logic trigger
assign trigger_in[8] = gpio_4; // user logic trigger
assign trigger_in[9] = gpio_7 // user logic trigger
```

- Connect the triggers to a wire called `trigger_in[9:0]`, as shown in the example.
- Connect the triggers to a user GSV clock stop trigger module as shown below:

```
ccm_gsv_clk_stop_user #(10) clk_stop_inst (trigger_in);
```

The number 10 defines the trigger input parameters, and the current maximum is 10 triggers.

The GSV clock stop trigger module is responsible for stopping all the CCM clocks to enable GSV capture. At runtime, use the clock module to select which of the 10 triggers to use to stop the clocks. You can also use it to combine different triggers by masking them with a MCAPIM register. By default, all triggers are ORed. For details on masking the triggers using ProtoCompiler Runtime, see [Masking Trigger Source Bits in Runtime, on page 790](#).

3. Run the GSV flow to generate the .ll file and bit files.
4. Start the ProtoCompiler Runtime and load the readback project from `exportResultsDir/system/readback/debug.prj`.

5. Load the \*.ll file, or \*.ba.ll file if the design is backannotated. Click the RR icon on the ProtoCompiler Runtime. Load files using their FPGA or TSS names:
  - To load mb1\_uA.ll in unified compile flow—mb1.uA
  - To load mb1\_uA.ll in standard compile BA flow—mb1\_uA\_ba.ll
  - To load the mb1\_uB.ll and mb1\_uB\_ba.ll—mb1.uB
6. Enable the desired stepping mode for readback. With the new clock generator flow, two stepping modes are available:
  - Driver clock readback (default)—Readback happens by stopping or stepping the driver clock. Use the following command to use this option:

```
package require CCM
CCM::open ../../
ccm stepping_mode -driver
```
  - CCM clock readback—Readback happens on the CCM generated clocks. Use the following command to use this option:

```
package require CCM
CCM::open ../../
ccm stepping_mode {-pos_cclock_id | -neg_cclock_id} [CCM::cclock_id
<clockID>]
```

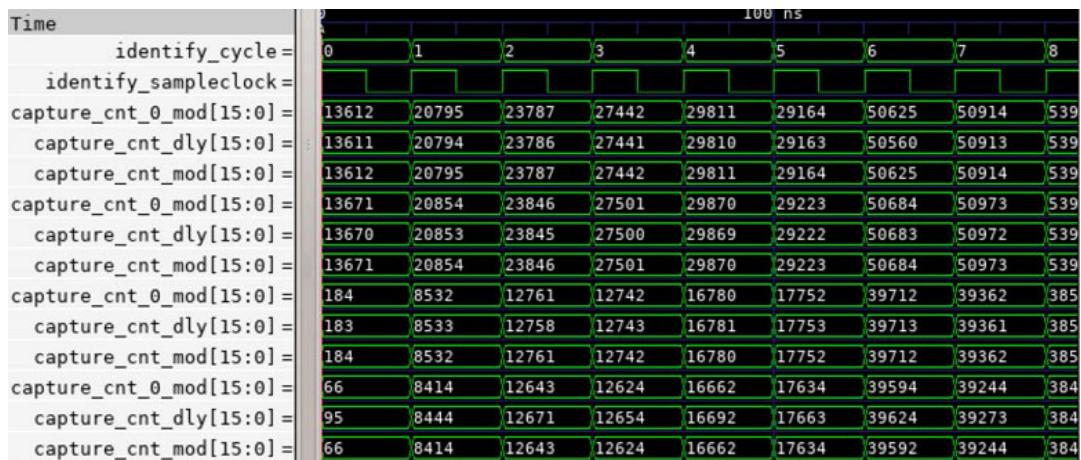
Details on CCM clocks in the design and its relative path is available in the zebu\_config.tcl file under the <exported runtime path>/system directory.

You can control whether the readback happens on the positive edge (-pos\_cclock\_id) or the negative edge (-neg\_cclock\_id) of the CCM clock. In the following example, the readback happens on the CCM generated clock (clk0) on positive edge:

```
ccm stepping_mode -pos_cclock_id [CCM::cclock_id cnt_demo_top.clk0]
```

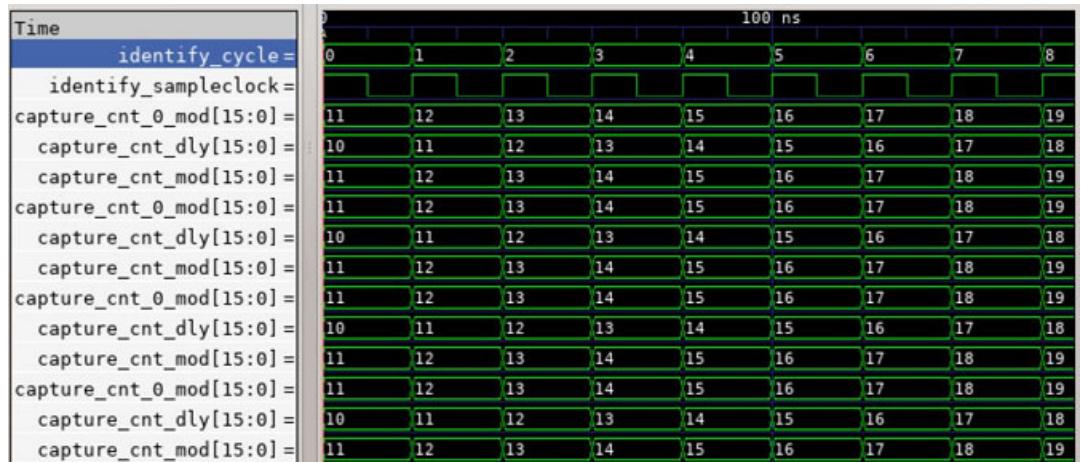
You can also enable or disable CCM clocks while performing the readback using the -uncontrolled option with the ccm clock\_mask command. By masking the clocks, you can make sure the readback happens only on CCM Clocks that are not masked. In the following example, the clock cnt\_demo\_top.clk0 is disabled or masked:

```
ccm clock_mask -cclock_id [CCM::cclock_id cnt_demo_top.clk0]
-uncontrolled 1
```

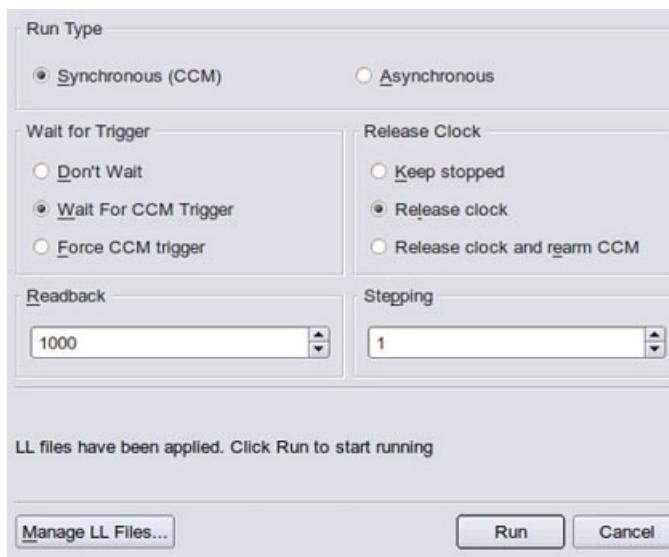


To unmask or enable the CCM clocks again, use the following command:

```
ccm clock_mask -cclock_id [CCM::cclock_id cnt_demo_top.clk0]
-uncontrolled 0
```



7. Use the regular runtime options in the GUI to stop clocks and perform readback by stepping the clocks.



8. Use the waveform streaming to save time on writing the FSDB. For details, see the [readback stream\\_waveform](#) command in the *HAPS Prototyping Debugging Environment Reference*.

## Masking Trigger Source Bits in Runtime

To mask bits of a trigger source during runtime, issue this command in the runtime shell to enable or disable individual bits:

```
ccm trigger -any -mask <32-bit_hex_value>
```

You can only write to bits [9:0] of the mask value. Each of the writable bits can be individually masked (1) or unmasked (0). The default is 0. For example: To mask the trigger source bits 0, 4, and 5, use this command:

```
ccm trigger -any -mask 110001
```

You can also use the GUI to enter this information:



The CCM block contains a register which can be accessed through Confpro. This register has two functions. The MSB bit defines the operation to run on the triggers: OR (0) or AND (1). Mask bits enable/disable specific trigger inputs. The default is 0, and defines how many triggers are active. If a trigger is to be disabled, mask it by setting it to 1.

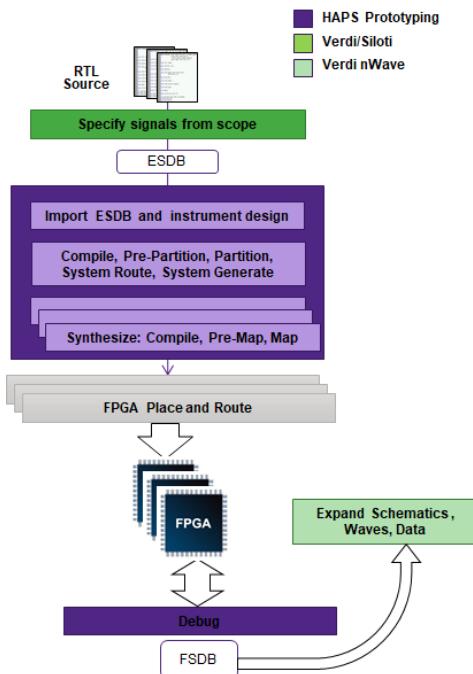
## Limitations

- CCM clock readback above 16 MHz is not recommended.
- GSV plus DPI flow does not support CCM clock readback.
- Clock generator for mixed system is supported only on HAPS-100 systems.
- Only OR combiner is supported for multiple trigger inputs to CCM.
- Phase shifting of clocks is not supported.

# Integrated Debugging with the Verdi Software

You can incorporate functionality from the Synopsys Verdi® automated debug system with its powerful data analysis capabilities into the HAPS ProtoCompiler debug flow. Verdi features include waveform viewing, comparison functionality, and waveform correlation to RTL. Use the Verdi capabilities in the prototyping debug flow to take full advantage of deep sample buffers. You can also combine the Verdi Siloti capabilities with the instrumentor to extract only the essential signals from a targeted part of the design for instrumentation and debugging.

The following figure illustrates the integrated Verdi flow. In it, Verdi/Siloti generates an Essential Signal Database (ESDB) which is imported into the instrumentor for signal instrumentation. The instrumented design is then synthesized, placed and routed, and programmed into the FPGA. The debugger samples the data and writes out a Fast Signal Database (FSDB) of the debugged signals, which is then displayed in the Verdi nWave viewer. You can then use the generated FSDB for data expansion and waveform viewing, and to compare and correlate the waveform to the RTL.



Note these points before you start using the flow:

- Integrated debugging with Verdi Software is supported only on Linux.
- HAPS-Verdi flow does not support SV interfaces.
- Do not reuse the correlation database (CRDB) generated using an earlier version of Verdi. Use the recommended version of Verdi to generate CRDB and to export DE.

For more detail, refer to these links:

- [Using Verdi with the Standard Compiler](#), on page 793
- [Using the Verdi Flow with a UC Design](#), on page 803
- [Launch Verdi](#), on page 794
- [Using Verdi and Siloti for Debugging](#), on page 797
- [Instrumenting Verdi Signals \(ESDB\)](#), on page 798
- [Generating the Fast Signal Database \(FSDB\)](#), on page 799
- [Debugging with the Waveform Viewer](#), on page 800

## Using Verdi with the Standard Compiler

Provides an automated integration of Verdi in ProtoCompiler and ProtoCompiler\_runtime for a seamless design debug. ProtoCompiler generates all the required scripts, and debug environment, along with correlation for the given design, which will provide a “push-button” approach for loading and debugging in Verdi.

With this approach, the verdi's data expansion can be performed using the FSDB that is generated from GSV flow using only the raw ll files, that is ll files without back annotation.

---

**Note:** Do not edit the scripts generated during the flow. The flow might fail if the scripts are edited.

---

## Implementation and Execution

Perform the following steps in ProtoCompiler to implement the design and export the automated Verdi debug environment:

1. Set the following options before executing the ProtoCompiler flow to enable the verdi integration:
  - option set verdi\_mode 1
  - option set prepare\_readback 1/option set prepare\_readback 2

---

**Note:** prepare\_readback 1 enables GSV and prepare\_readback 2 enables eGSV.

---

2. Execute the following export runtime command to create the verdi database.

```
export runtime -path <runtime path>
```

The command creates a verdi sub folder that contains all the required scripts.

3. Use the regular design flow to implement the design with or without instrumenting the design.

- Backannotation of the ll files has to be by-passed to avoid the creation of backannotated ll files. This can be performed by not using database apply\_state -backannotate <value>. If backannotation cannot be by-passed, then user must pick the raw ll files from the respective Place and Route folder(s) during the GSV execution in runtime.

For information on how to open Verdi, see [Launch Verdi, on page 794](#).

## Launch Verdi

The following steps describe how to launch Verdi:

1. Load the original .ll file from Vivado.
2. Once the samples are downloaded, generate the FSDB file using the following command:

```
write fsdb <filename> -iice <value>
```

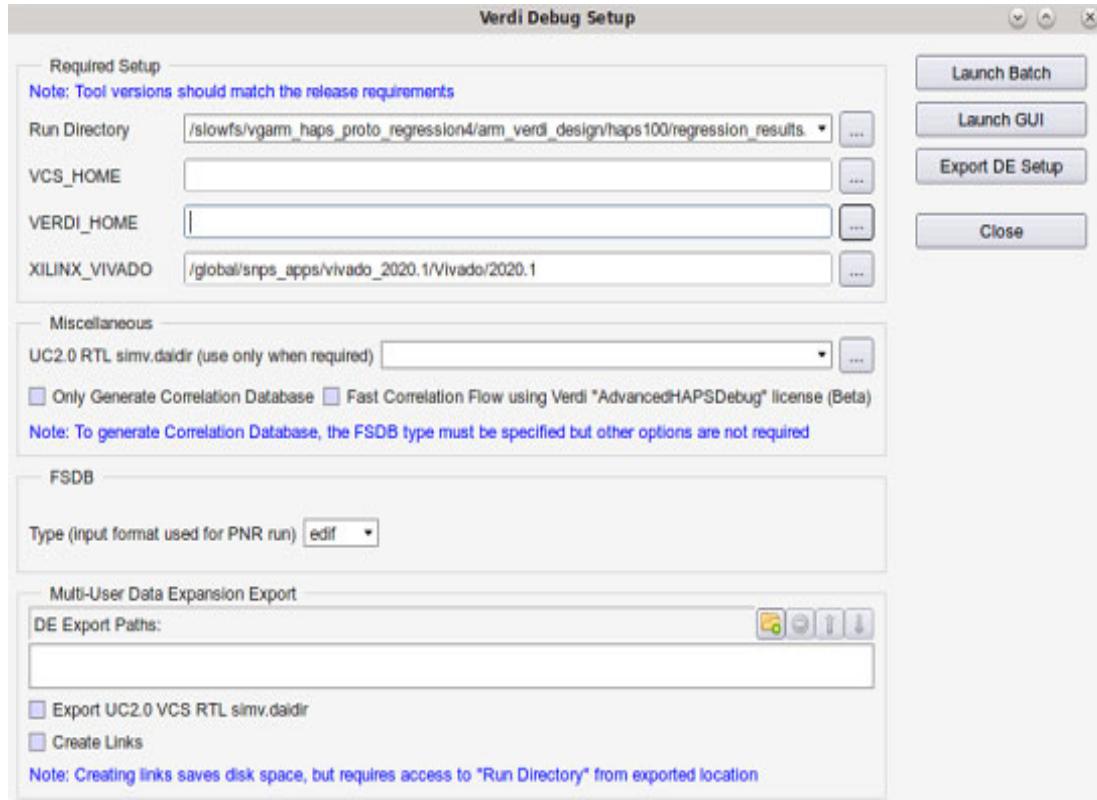
- Once the FSDB file is generated, launch Verdi using the following command:

```
launch verdi -run_dir <> -fsdb <> -fsdbtop <> -fsdbtype pnr
-gui
```

See `launch verdi` in the *HAPS Prototyping Command Reference* manual and `launch_verdi` command in the HAPS Prototyping Debugging Environment Reference manual.

The following steps describe how to launch Verdi through GUI:

- From the Debugger menu, select Verdi Debug. The Verdi Debug Setup window is displayed as shown below.



- Ensure that VCS\_HOME, VERDI\_HOME, and XILINX\_VIVADO are set. Also, if the unified compile results (VCS simv.dadir) are from a different

location, specify the same. By default, the tool locates from the current database.

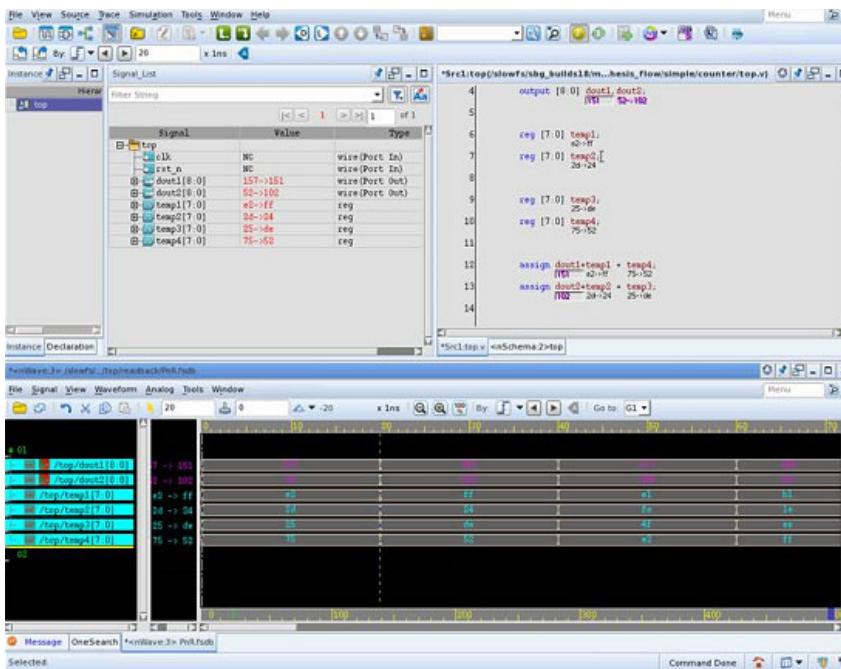
### 3. Under FSDB:

- Path - Provide the Path to the generated FSDB file.
  - Top Module - Enter the name of the top module of the design.
  - Type - By default, edif is set. Allowed values are edif and verilog.

4. Click Launch Batch to open Verdi in Batch mode.

5. Click Launch GUI to invoke Verdi GUI and run the flow.

- After performing the `launch_verdi` command, the verdi GUI will automatically display the data expansion on the RTL signals as shown below.



## Using Verdi and Siloti for Debugging

The following procedure describes the steps illustrated in the diagram in [Integrated Debugging with the Verdi Software, on page 792](#) for Standard Compile.

### 1. Set up the project.

- Connect the HAPS hardware system to a Linux machine.
- Ensure that the Verdi software is installed on a Linux machine.
- Configure the Verdi license as described below:

The tool searches for licenses in this order: SNPSLMD\_LICENSE\_FILE,

#### **Binary File**

```
setenv VERDI_HOME PathToNovasInstallation
set path = ($VERDI_HOME/bin $path)
```

#### **License File**

```
setenv NOVAS_LICENSE_FILE license_file:$NOVAS_LICENSE_FILE
Alternatively use LM_LICENSE_FILE or SNPSLMD_LICENSE_FILE
```

NOVAS\_LICENSE\_FILE, LM\_LICENSE\_FILE

### 2. Generate the Essential Signals Database (ESDB).

The ESDB comprises the essential set of signals you want to capture for debug visibility into the block or area of interest.

- Load the RTL design in the Verdi tool.
- Run behavioral analysis in the Verdi software. Use the visibility analysis engine to determine the essential set of signals you need for later processing with the data expansion engine.
- Generate ESDB by specifying this command: run\_generate\_esdb

You can also use the vericom and esa commands to specify the working scope and the directory. This example generates the ESDB for the specified working scope, which is the top module P\_Idt\_Struct\_tbtop, and saves it in a directory called es.esdb++. The essential signal database is called es.

```
vericom -2009 -sverilog +v2k -lib work -f files.f -incdir ../../src +incdir+../../src
```

```
esa -lib work -top P_Iddt_Struct_tbtop -db es -bas P_Iddt_Struct_tbtop
-libPath work -bdb_file work
```

3. Start the prototyping software and instrument the ESDB signals.  
See [Instrumenting Verdi Signals \(ESDB\), on page 798](#) for details.
4. Run through the rest of synthesis, placement and routing; generate the bit file, and program the FPGA as usual.
  - Make sure that the HAPS system you are using has been configured with the Confpro utility.
  - Use the generated bit file to program the FPGA on the HAPS system.
5. Launch HAPS ProtoCompiler runtime, and run the debugger to sample the data and generate the fast signal database.  
See [Generating the Fast Signal Database \(FSDB\), on page 799](#) for details.
6. Export files for Verdi debugging.  
See [Exporting Files for Verdi, on page 800](#) for details.
7. Use the Verdi waveform viewer to debug the design.  
See [Debugging with the Waveform Viewer, on page 800](#) for details.

## Instrumenting Verdi Signals (ESDB)

You can import signals from a Verdi database directly into the instrumentor by following the steps below.

1. Generate the essential signal database (ESDB) as described in [Using Verdi and Siloti for Debugging, on page 797](#)
2. Start the prototyping tool, and instrument the design (run pre\_instrument and edit idc -mode commands).
3. Parse the essential signal list from the ESDB using this command:

```
verdi getsignals ESDBpath
```

*ESDBpath* is the location of the es.esdb++ directory created previously.  
For example: verdi getsignals myDir/es

This command extracts the signals from the ESDB.

4. Instrument the essential signal list using the `verdi instrument` command.

This command instruments the essential signals. The signals are automatically instrumented as sample and trigger.

5. Instrument the sample clock.

Consult the built-in instrumentor documentation for details about instrumenting and configuring IICE.

6. Configure the IICE and instrument the design.

You can now return to the procedure described in [\*Using Verdi and Siloti for Debugging, on page 797\*](#), and complete the FPGA implementation and generate the bit file.

## Generating the Fast Signal Database (FSDB)

Once you have instrumented the Verdi database and implemented the FPGA, use the debugger to sample the data and generate the fast signal database (FSDB).

To generate the FSDB database, follow this procedure:

1. Instrument the Verdi database and implement the FPGA as described in [\*Using Verdi and Siloti for Debugging, on page 797\*](#).
  - Make sure that the HAPS system you are using has been configured with the Confpro utility.
  - Use the generated bit file to program the FPGAs on the HAPS system.
2. Start the debugger with the `protocompiler_runtime` command.
3. In the Debugger Preferences dialog box set Synopsys Verdi nWave as the default waveform viewer.
4. Arm the trigger conditions and click Run to download the sample buffer.
5. Generate the fast signal database using the following command syntax:

```
write fsdb -iice iiceID
```

After the trigger occurs, the tool generates an FSDB file for the captured samples of the essential signals, and saves it in the current working directory. The next step is to export the files (see [\*Exporting Files for\*](#)

[Verdi](#), on page 800) so that you can display the results in the Verdi nWave viewer.

## Exporting Files for Verdi

1. Generate an FSDB database.

See [Generating the Fast Signal Database \(FSDB\)](#), on page 799 for details.

2. Set the verdi\_mode option to 1.
3. Run the export runtime command and specify the FSDB database.

The export runtime command exports all scripts, setup and correlation data needed to run Verdi. For the complete syntax, see [export runtime](#), on page 58 in the Command Reference.

This is an example of the commands used:

```
option set verdi_mode 1
run compile -filelist addFilessynth.txt
run pre_map
run map
export runtime -path fsdbFilename
launch verdi -fsdb fsdbFilename
```

4. After exporting the files, you can launch Verdi and use the waveform viewer for debugging.

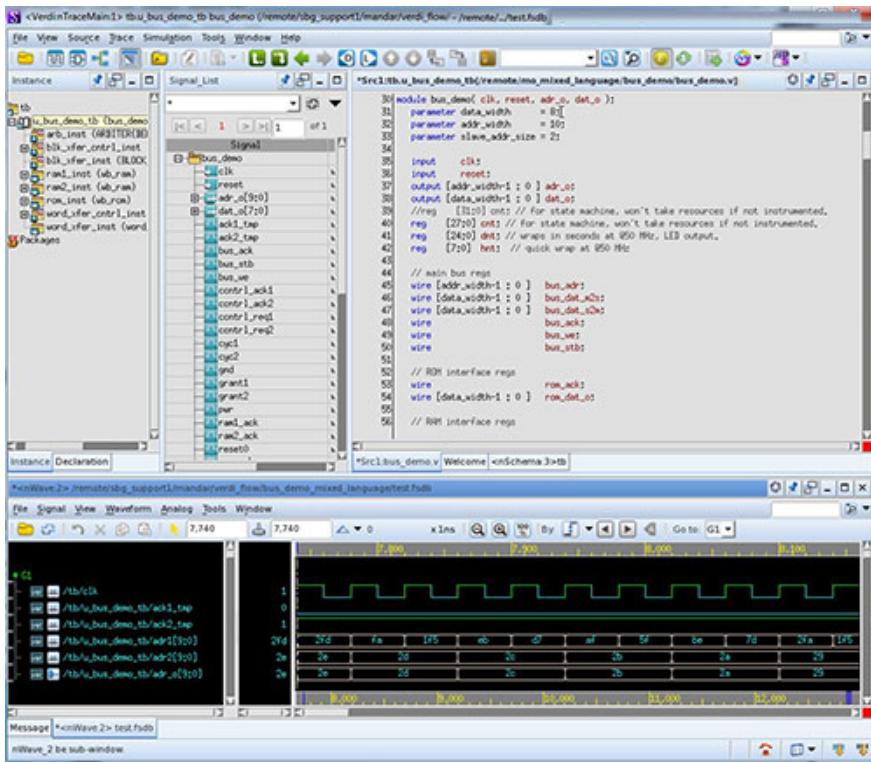
For details, see [Debugging with the Waveform Viewer](#), on page 800.

## Debugging with the Waveform Viewer

You can use the Verdi nWave waveform viewer for data expansion and debugging once you have generated the FSDB.

1. Export the files, as described in [Exporting Files for Verdi](#), on page 800.
2. Launch the Verdi software from the shell.

```
launch_verdi -fsdb Filename
```



3. Open the nWave viewer and view signals that are not in the FSDB with the Data Expansion engine.
  - Click the New Waveform toolbar icon ()
  - Click the Get Signals icon ()
  - In the Get Signals hierarchy browser pane, click the + icon to expand items. The signals in bold are signals that are not in the FSDB, and which can be expanded (calculated on the fly) by the Data Expansion engine.
  - Select the signals you want to view in the waveform viewer and click OK to add them to the waveform. Drag left to select several signals; make sure to include the top-level I/O port.

The values for all signals are now annotated. The ones in purple are expanded signals which did not come from the FSDB file, but were calculated. This is an example of the results:

```
344 ase (Struct_P_LDDT_iport.iport_enum_member)
345 ADD
346 ADD : begin//(
347 //Struct_P_LDDT_oport.oport_P_LDDT_struct_member <= Struct_P_LDDT_InsideMem
348 Struct_P_LDDT_oport.oport_P_lddt_Struct_reg_member <= Struct_P_LDDT_iport_9
349 a
350 Struct_P_LDDT_oport.oport_P_lddt_Struct_logic_member <= Struct_P_LDDT_iport_8
351 a
352 Struct_P_LDDT_oport.oport_P_lddt_Struct_bit_member <= Struct_P_LDDT_iport_7
353 b
354 Struct_P_LDDT_oport.oport_P_lddt_Struct_byte_member <= Struct_P_LDDT_iport_6
355 f8
356 a
357 Struct_P_LDDT_oport.oport_P_lddt_Struct_shortint_member <= Struct_P_LDDT_iport_5
358 fffd
359 mbb
360 Struct_P_LDDT_oport.oport_P_lddt_Struct_longint_member <= Struct_P_LDDT_iport_4
361 ffffffff
362 mmffff
363 Struct_P_LDDT_oport.oport_P_lddt_Struct_int_member <= Struct_P_LDDT_iport_3
364 1
365 mmuu
366 Struct_P_LDDT_oport.oport_P_lddt_Struct_integer_member <= Struct_P_LDDT_iport_2
367 3
368 mmme
369 Struct_P_LDDT_oport.oport_P_lddt_Struct_SignedReg_member <= Struct_P_LDDT_iport_1
370 b
371 a
372 Struct_P_LDDT_oport.oport_P_lddt_Struct_SignedLogic_member <= Struct_P_LDDT_iport_0
373 8
374 a
375 Struct_P_LDDT_oport.oport_P_lddt_Struct_SignedBit_member <= Struct_P_LDDT_iport_9
376 c
377 a
378 Struct_P_LDDT_oport.oport_P_lddt_Struct_UnsignedByte_member <= Struct_P_LDDT_iport_42
379 43
```

# Using the Verdi Flow with a UC Design

The Verdi software is integrated with the prototyping software to provide a seamless design debug environment to validate UC designs. The prototyping tool generates the required scripts, and debug environment, which provides a "push-button" way to load the design and debug with the Verdi viewer.

Use the Verdi tool in conjunction with the Global State Visibility (GSV) functionality. GSV leverages the Xilinx readback feature and provides a snapshot of all registers in a configured design at a particular clock cycle, thus offering full visibility. It does not require instrumentation and is instantly available. There are two GSV modes: legacy GSV and enhanced GSV (eGSV). The eGSV mode is implemented only when the design is instrumented but there are no other changes in the use model.

With this approach, you can perform Verdi data expansion using the FSDB that is generated from the GSV flow using the Vivado .ll files.

See these topics:

- [Running the Verdi Flow For Designs with GSV](#), on page 803
- [Running the Verdi Flow For Designs with Instrumented/Probe Signals](#), on page 804

## Running the Verdi Flow For Designs with GSV

The steps below describe how to implement the design and export the Verdi debug environment for designs with GSV.

1. Set the following options before executing the synthesis or prototyping design flow:

```
option set verdi_mode 1
option set prepare_readback 1 | 2
```

The readback option is only required for debug with GSV. Set `prepare_readback` to 1 for GSV, and to 2 for eGSV. Setting this option generates the logic location (.ll) files when the bit files are generated. The .ll file contains mapped register information from the FPGA to design signal locations, and GSV uses these signal information.

2. Compile the design using the Unified Compile flow (launch uc and run compile).

3. Continue with the design flow through place and route.

Select the logic location (.ll) files generated by Vivado from the respective place-and-route folders when the GSV operation runs during runtime (see step 5). You can then launch Verdi from runtime, check correlation in the Verdi-generated crdb\_summary.log file and monitor the data expansion results.

4. Run the export runtime command to create the Verdi database.

```
export_runtime
```

The command creates a readback folder and a verdi folder that contains all the scripts required to run the Verdi.

5. Run debug and capture GSV.

- Start the debugger and launch the Verdi software. Details are described in [Launching the Verdi Tool, on page 805](#).
- Open the readback project.
- Click the Readback icon ( ) to open the dialog box. Load the GSV IICE.
- Load the .ll files, using the Readback IICE dialog box. FPGA names must follow the naming convention for individual FPGAs.
- Click Run and capture the GSV in an FSDB file.

```
write fsdb <fsdbName>
```

Make sure not to edit or modify scripts generated during the flow.

## Running the Verdi Flow For Designs with Instrumented/Probe Signals

You can export instrumented or non-instrumented designs to the Verdi environment for debug. The steps below describe how to implement the design and export the Verdi debug environment.

1. Set the following option before executing the synthesis or prototyping design flow:

```
option set verdi_mode 1
```

2. Compile the design using the Unified Compile flow (launch uc and run compile).

- Identify the essential signals using the `report debug_essential_signals` command ([report debug\\_essential\\_signals, on page 102](#) in the *Command Reference*). Include the signals in an IDC file, so that they are visible to the Verdi tool.
  - Continue with the regular design flow through place and route, with or without instrumenting the design.
3. Run the `export runtime` command to create the Verdi database.

```
export_runtime
```

The command creates a `verdi` folder that contains all the scripts required to run the Verdi.

4. Run debug with Verdi.
- Start the debugger.
  - Enable the trigger condition for GSV capture.
  - Run Readback capture.
  - Capture the FSDB waveform.

```
launch verdi -fsdb
```

- Check the Verdi-generated `crdb_summary.log` file for correlation.
- Monitor the data expansion results.

## Launching the Verdi Tool

The following steps describe how to launch the Verdi tool from the command line and from the GUI.

### Launching Verdi Through Commands

1. Make sure the environment variables for `VCS_HOME`, `VERDI_HOME` and `XILINX_VIVADO` are set correctly.
2. After the FSDB file is generated, start the Verdi software with the `launch verdi` command specifying the EDIF or Verilog flow, according to what was used for place and route.

EDIF example:

```
launch_verdi -run_dir debug_data -fsdb top.fsdb -fsdbtop top
-fsdbtype edif
```

### Verilog example:

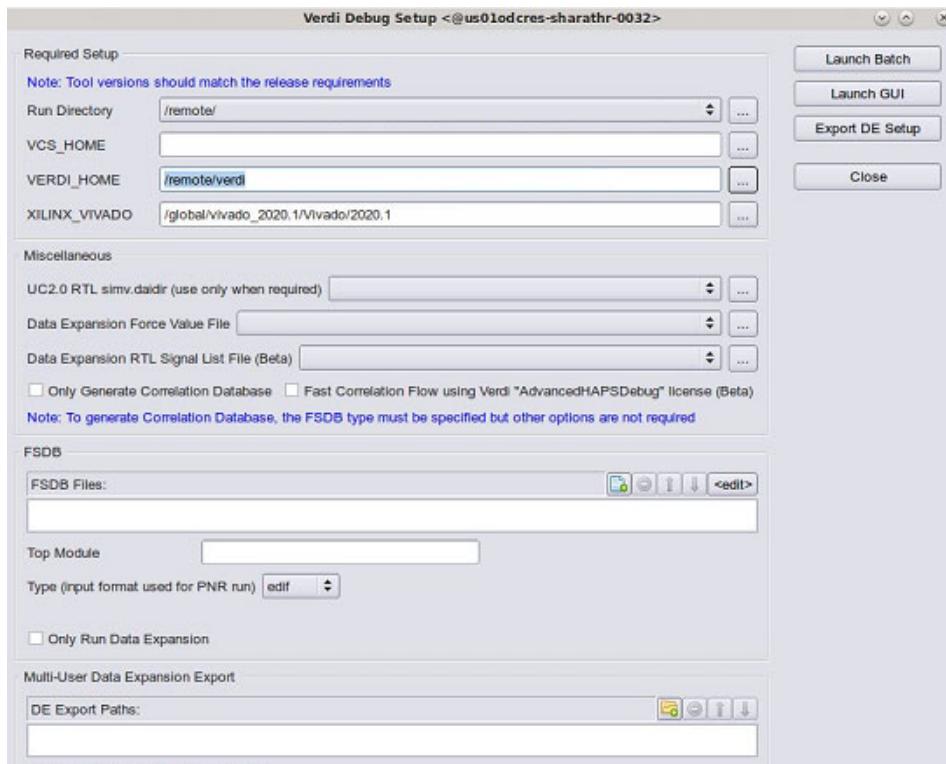
```
launch_verdi -run_dir debug_data -fsdb top.fsdb -fsdbtop top
-fsdbtype verilog
```

See [launch verdi, on page 76](#) in the *Command Reference* for syntax information.

## Launching Verdi from the ProtoCompiler Runtime GUI

1. Open the Verdi tool and select Verdi debug from the Debugger menu.

The Verdi Debug Setup window is displayed as shown below.

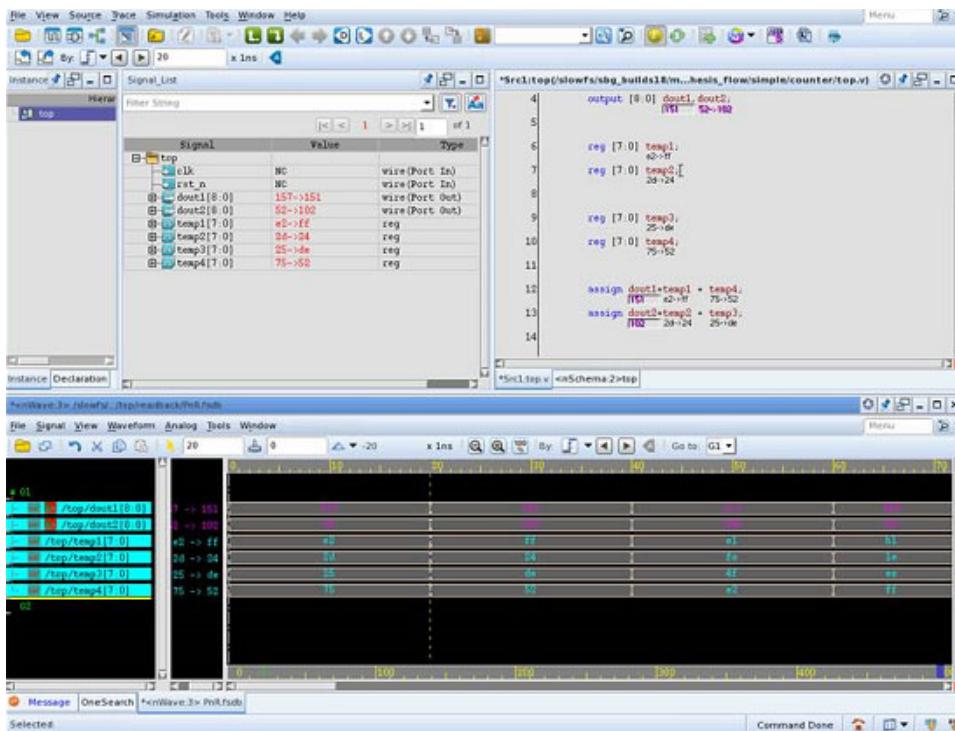


2. Check that the environment variables for Verdi are set correctly.

If **VERDI\_HOME** and **XILINX\_VIVADO** are not set correctly, these fields will be blank. By default, the tool searches for the UC results (VCS simv.daidir) in the current database. It is recommended that you do not override this default database that is generated by the tool.

3. Define the following in the FSDB section of the dialog box:
  - Path: Provide the path to the generated FSDB file.
  - Top Module: Enter the name of the top module of the design.
  - Type: The netlist type generated by the ProtoCompiler tool. Set this to edif or verilog, based on the option used.  
Example: If you have used the command option set write\_verilog 1, then use launch\_verdi -run\_dir ..\verdi -fsdbtop <topmodule> -fsdbtype verilog
4. Click Export DE to generate folders that allows multiple users to perform data expansion from the same design correlation database. The tool creates a folder in the user specified path and copies CRDB along with necessary scripts to launch data expansion. See [\*export\\_verdi\\_de\*, on page 59](#) in the *Command Reference* for command line syntax information.
5. Click the appropriate button to start the tool.
  - Click Launch Batch to open the Verdi tool in batch mode and run the flow.
  - Click Launch GUI to start the Verdi GUI and run the flow.

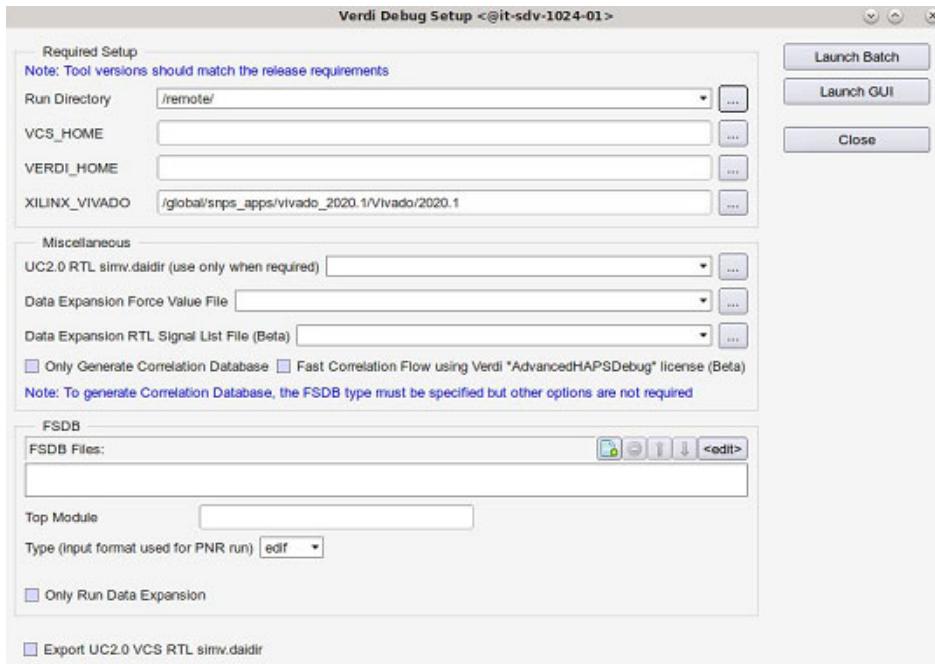
The Verdi GUI opens and automatically displays the data expansion for the RTL signals, as shown below.



## Launching Verdi from the ProtoCompiler GUI

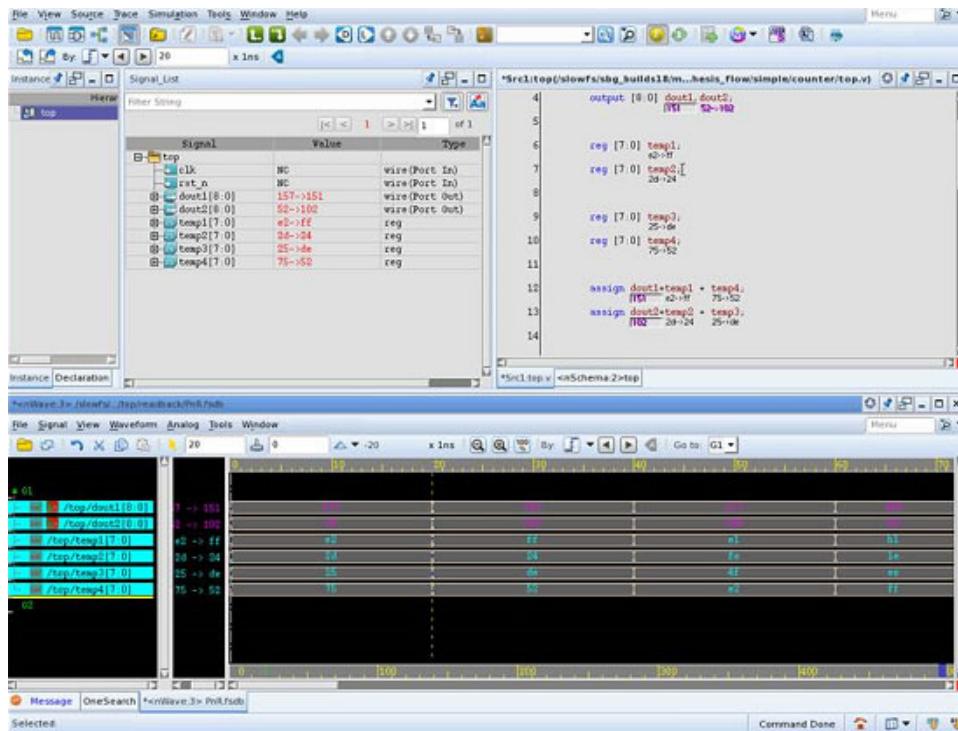
1. Open the ProtoCompiler tool and select State > Verdi debug from the menu.

The Verdi Debug Setup window is displayed as shown below.



2. Check that the environment variables for Verdi are set correctly.  
If VERDI\_HOME and XILINX\_VIVADO are not set correctly, these fields will be blank. By default, the tool searches for the UC results (VCS simv.daidir) in the current database. It is recommended that you do not override this default database that is generated by the tool.
3. Define the following in the FSDB section of the dialog box:
  - Path: Provide the path to the generated FSDB file.
  - Top Module: Enter the name of the top module of the design.
  - Type: The netlist type generated by the ProtoCompiler tool. Set this to edif or verilog, based on the option used.  
Example: If you have used the command option set write\_verilog 1, then use launch verdi -run\_dir ..//verdi -fsdbtop <topmodule> -fsdbtype verilog
4. Click the appropriate button to start the tool.
  - Click Launch Batch to open the Verdi tool in batch mode and run the flow
  - Click Launch GUI to start the Verdi GUI and run the flow.

The Verdi GUI opens and automatically displays the data expansion for the RTL signals, as shown below.



# Running Formal Verification with Checkpoints

This verification flow is based on the creation of check points, which can be compared to the original Verilog using the Formality or VC Formal (VCF) tool to run formal verification. Refer to the Formality and VCF documentation for details on how to use those tools.

## Verifying with Checkpoints

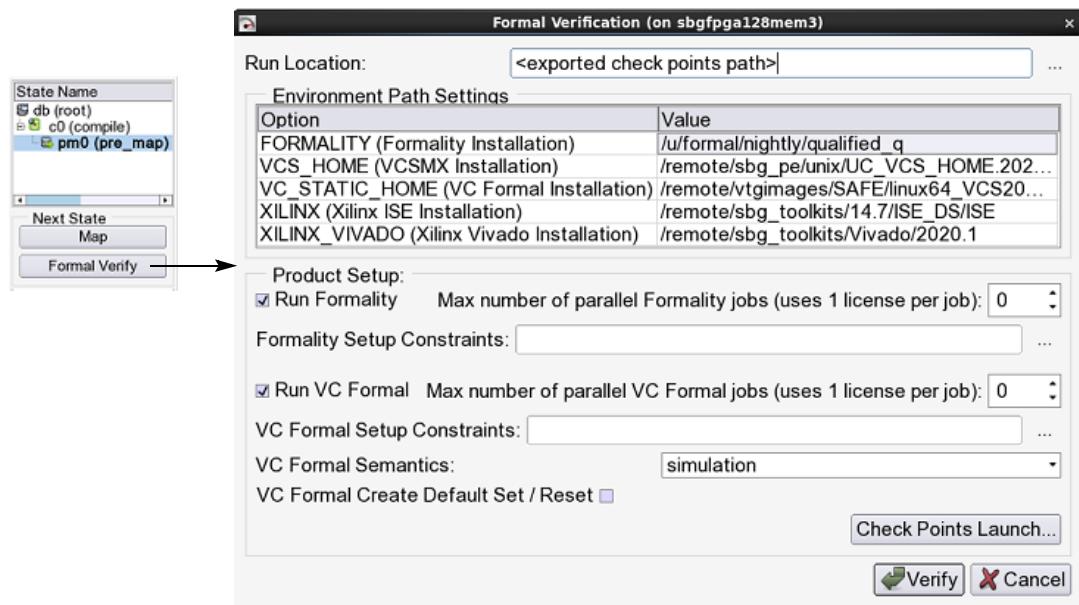
1. Set environment variables for the verification tools:
  - Set the Formality variable (FORMALITY).
  - Set the VCS variable (VCS\_HOME).
  - Set the VCS Static variable (VCS\_STATIC\_HOME).
  - Set the Xilinx Vivado installation (XILINX\_VIVADO).
2. Set verification options from the GUI or the command line:
  - Enable verification mode: option set verification\_mode 1. In the GUI, set Verification Mode.
  - Enable the creation of checkpoints: option set verify\_check\_points 1. In the GUI, set Verification Check Points. When this option is enabled, the tool automatically creates checkpoints based on the design modules (SCMs).

After the compile stage, the tool writes out checkpoints for the modules. Each verification check point contains the following data that the verification tools can use:

- Complete environment setup, formal tool setup, and launching data
- Complete reference and implementation Verilog designs, and other data
- Formal tool constraints such as clocks, black boxes etc.
- Guidance and matching data for achieving maximum formal convergence
- Interface for controlling various capabilities of specific formal tools based on the check point requirements
- Report generation mechanisms

- Handshaking mechanisms between the prototyping and verification tools
3. Export files for formal verification.
    - Go to a valid database state, such as compile or pre-map. For a list of the database states, refer to [export verification, on page 60](#).
    - Run the export verification command.

```
export verification -check_points -path dir
```
  4. Open the dialog box for formal verification options.
    - From the pre-map or other valid stage in the GUI, click Formal Verify, as shown in the next figure below.
    - To use the command line, specify the formal\_verify command from a valid database state. For the syntax of this command, see [formal\\_verify, on page 62](#).



5. Specify options for, and run formal verification with checkpoints.
  - Enable the verification tool to use for the formal verification run. You can choose Formality, or VC Formal, or both. See [Selecting a](#)

[Verification Tool for Checkpoint Analysis, on page 813](#) for some guidelines.

- Specify options for the verification tool. For Formality, specify the number of parallel jobs and a file with setup constraints. For VCF, specify the maximum number of parallel jobs, setup constraints file, default set/reset, and semantics mode. See [Selecting a Verification Tool for Checkpoint Analysis, on page 813](#) for more information.
  - Click Check Points Launch. This runs verification and opens a window with the results.
6. Analyze results in this window.

See [Analyzing Checkpoint Verification Results, on page 815](#).

## Selecting a Verification Tool for Checkpoint Analysis

Both the VC Formal and Formality tools can be used for checkpoint analysis. The table compares some criteria for the two tools.

| VCF                                                                                                             | Formality                                                                                 |
|-----------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|
| Use for all sequential equivalence checking, especially for GCC.                                                | Use for combinational equivalence checking                                                |
| Use for global prep, because it can handle cross-group, cross-iteration optimizations for distributed synthesis | Not as well suited for global prep, because it does not handle cross-group optimizations. |
| May require lot of machine resources for formal convergence                                                     | May require machines with larger memory; possible long runtimes                           |

- VCF Options
  - Specify the number of formal workers for each VC Formal run. The recommendation is to specify a minimum of 50 workers for GCC verification. Specify an optional setup constraint file with additional information. This is an example of typical commands included in this file:
 

```
set_grid_usage -type rsh=50
set_fml_var fml_max_time 24H
set_fml_var fml_max_mem 64GB
```
  - There are no default sets and resets. This could lead to formal failures in some cases. Some designs might require that a default set/reset be

defined, and that all design inputs other than clocks be treated as free inputs for verification.

- Define formal semantics correctly. In some cases, default simulation semantics could cause failures, because the formal behavior might try to match the simulation behavior for ProtoCompiler output. In such cases, rerun the failed jobs using synthesis or auto semantics. The former would better match the ProtoCompiler synthesis output, while auto runs the simulation semantics followed by synthesis semantics on failed assertions.
- Formality Options
  - Specify additional information for machine configurations in an optional setup constraints file. The file contains commands like the following:

```
set_host_usage -max_cores 8
```

## Examples

### Verification Check Points After Compile

```
database load db -autocreate -technology HAPS-80
option set verification_mode 1
option set verify_check_points 1
launch uc...
run compile...
export verification -check_points -path check_points_data
set fvId [formal_verify -path check_points_data -post_compile...]
run pre_map
run map
job wait $fvId
```

### Verification Check Points After Pre-Map

```
database load db -autocreate -technology HAPS-80
option set verification_mode 1
option set verify_check_points 1
launch uc...
run compile...
run pre_map...
export verification -check_points -path check_points_data
set fvId [formal_verify -path check_points_data -post_premap...]
run map
job wait $fvId
```

## Analyzing Checkpoint Verification Results

1. Set up options and run checkpoint verification as described in [Verifying with Checkpoints, on page 811](#).
2. Analyze results in the results window that opens.

The following figure shows the window when verification was run from the pre-map stage; the window after compile is similar, except for the State at the bottom left.

The screenshot shows a software interface for managing verification processes. At the top, there's a table titled "Process" with columns for "State", "Run Time", "Product", "Report", and "Verification Status". Below the table is a "Deselect All" checkbox and a "Run Location" dropdown set to "<exported check points path>". Underneath are two groups of controls: "Product: Max Parallel Jobs:" with radio buttons for "VC Formal" (selected) and "Formality" (unchecked), and "VC Formal Semantics: simulation" with a dropdown menu. To the right of these is a note about launching verification from a run location. At the bottom are buttons for "Start" and "State: Pre-Map".

| Process                                             | State    | Run Time | Product   | Report                                                  | Verification Status |
|-----------------------------------------------------|----------|----------|-----------|---------------------------------------------------------|---------------------|
| formal_verify_flow (p)                              | -        | 00:00:00 |           |                                                         |                     |
| before_sggcc_vs_after_sggcc (p)                     | -        | 00:00:00 |           |                                                         |                     |
| ..._after_sggcc:before_sggcc_vs_after_sggcc-runvcf  | Error    | 00:00:52 | VC Formal | <a href="#">Status View</a><br><a href="#">Log File</a> | FAILED              |
| iter_1_top_glob_begin_vs_iter_1_top_glob_end (p)    | -        | 00:00:00 |           |                                                         |                     |
| ..._1_top_glob_begin_vs_iter_1_top_glob_end-runvcf  | Error    | 00:00:43 | VC Formal | <a href="#">Status View</a><br><a href="#">Log File</a> | FAILED              |
| ...p_glob_begin_vs_iter_1_top_glob_end-runformality | Error    | 00:00:04 | Formality | <a href="#">Log File</a>                                | FAILED              |
| iter_3_top_glob_begin_vs_iter_3_top_glob_end (p)    | -        | 00:00:00 |           |                                                         |                     |
| ..._3_top_glob_begin_vs_iter_3_top_glob_end-runvcf  | Canceled | 00:00:04 | VC Formal | <a href="#">Status View</a><br><a href="#">Log File</a> |                     |
| ...p_glob_begin_vs_iter_3_top_glob_end-runformality | Complete | 00:00:04 | Formality | <a href="#">Log File</a>                                | SUCCEEDED           |

Deselect All

Run Location: <exported check points path>

Product: Max Parallel Jobs:  VC Formal 5  Formality 5

VC Formal Semantics: simulation

VC Formal Create Default Set / Reset

Launch Formality and VC Formal verification from the given run location. Run location is expected to contain the appropriate formal verification data and setup

State:  Pre-Map Remote Launch Command:  Start

The Process column shows all the checkpoints in the design that you can selectively run, and the state of the run. If you specified both VCF and Formality, the jobs are monitored and canceled in whichever tool successfully finishes verification first.

3. Check for failures.
  - Check the State column for status.
  - If a job is failing, check the links in the Reports column. The Status View link shows the complete formal log where you can check details of the run.

The following figure shows the failing assertions for a check point.

The screenshot displays the 'VC Formal Status' tab of the Verdi Flow interface. It includes two main sections: 'VC Formal Information' and 'Check Point VC Formal Verification Summary (Status : FAILED)'.

**VC Formal Information:**

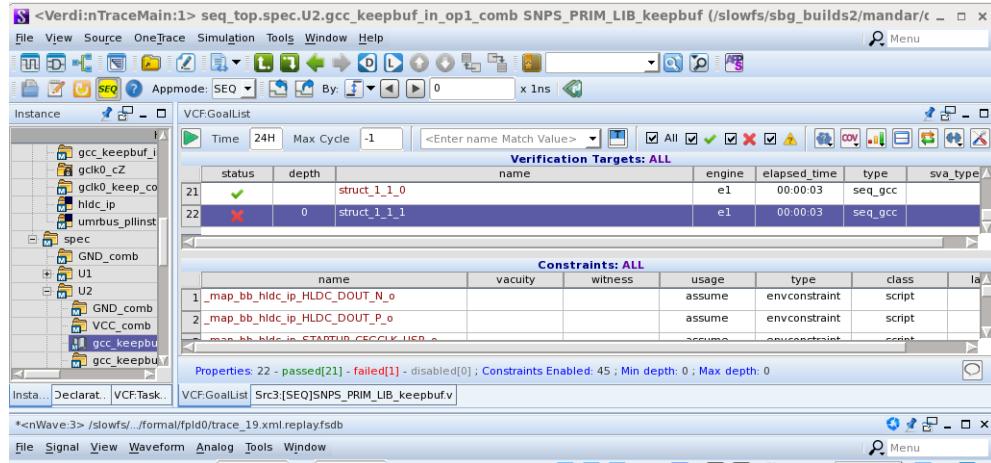
- Notes:**
  - Set the appropriate VC\_STATIC\_HOME environment variable before running verification
  - Runs check point netlist verification in the exported directory
  - Verification failure may not always be a logic issue
  - The generated scripts and files should not be modified
- Path:** /remote/e/vtg/images/SAFE/linux64\_VCS2020.03/release-build/vc\_static/bin/vcf
- Run Directory:** /slowfs/sbg\_builds2/madar/test/test\_check\_point/gen\_task\_25\_0314/check\_points\_verif\_2/top/iter\_1\_top\_glob\_begin\_vs\_iter\_1\_top\_glob\_end/post\_premap
- Guidelines:** readme
- Verification Black Boxes:** iter\_1\_top\_glob\_begin\_vs\_iter\_1\_top\_glob\_end\_verif\_bbbox.vcf.vy
- Assertions Summary:** assertions\_summary
- Status:** Finished in 40.31 seconds

**Check Point VC Formal Verification Summary (Status : FAILED):**

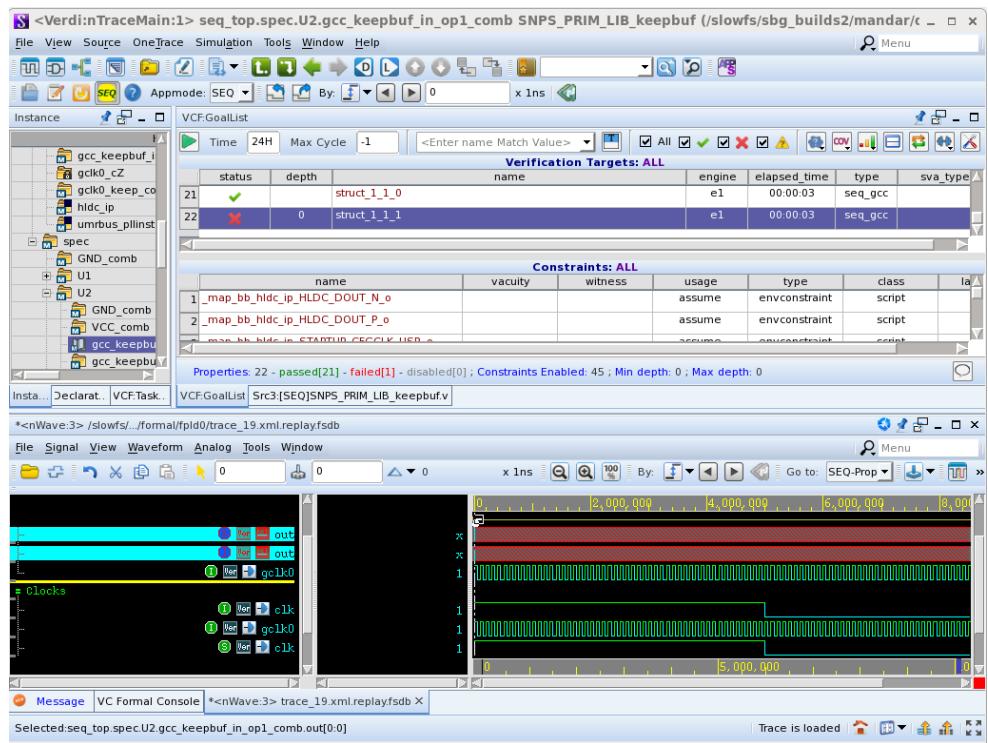
| Design | Verification Black Box | Status | Assertions passing vs. total<br>(34 / 36) | Report      | Session              |
|--------|------------------------|--------|-------------------------------------------|-------------|----------------------|
| top    | —                      | Failed | 34 / 36                                   | top_vcf.log | <a href="#">open</a> |

[Detailed Verification Report](#)

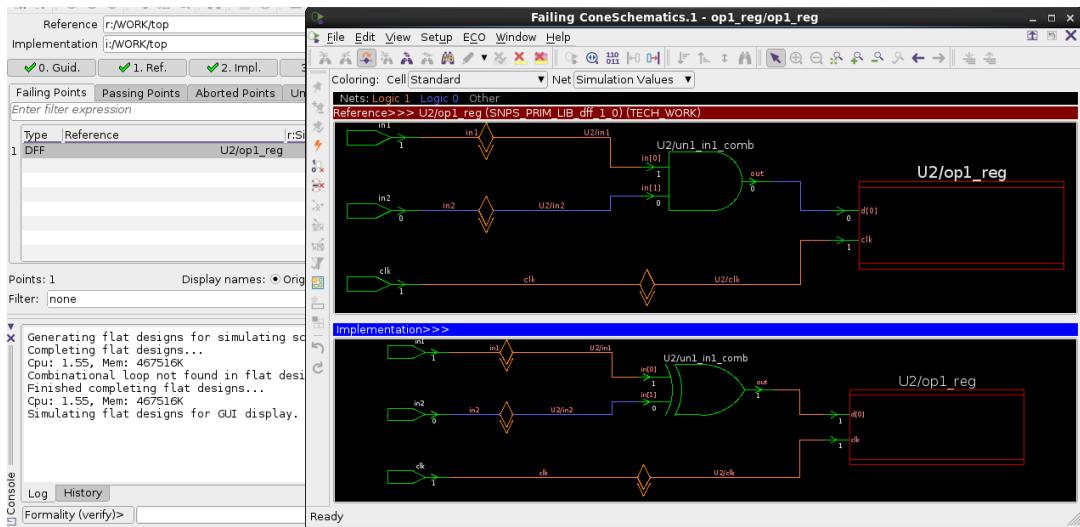
- Click the open link (under Session on the lower right) to open the corresponding VCF or Formality session.



- Analyze the results in the VCF session that opens.



If verification was run with the Formality software, that interface opens instead of the VCF one:



For details about using these tools to analyze and verify results, refer to the documentation for the Formality and VC Formal products.

## CHAPTER 7

# Migrating Designs for Synthesis

---

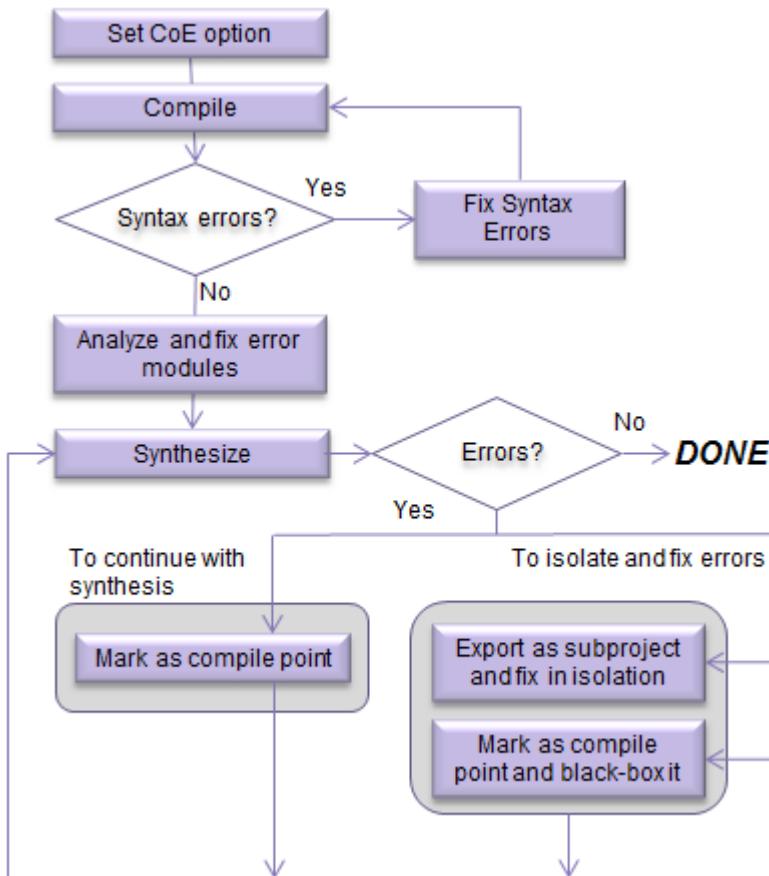
The tool includes certain features that facilitate the migration of ASIC designs and to improve productivity as you implement the design. These topics describe them in more detail.

- [Using Continue on Error](#), on page 820
- [Including UPF Specifications](#), on page 827
- [Converting Gated Clocks](#), on page 862
- [Editing Netlists](#), on page 907
- [Using Cross-Module Referencing \(XMR\)](#), on page 913
- [Converting Designs for Unified Compile](#), on page 920

# Using Continue on Error

The Continue on Error (CoE) feature significantly reduces overall synthesis runtime by reducing the number of synthesis iterations. This can be a significant advantage when prototyping or handling large designs.

By default, the compiler stops the compilation process as soon as it encounters an error in the design. The Continue on Error option allows the compilation process to continue for certain non-syntax compiler errors. As a result, multiple problems might require multiple iterations to identify and correct the issues. This translates to long turn-around times, especially for complex designs with lengthy compilation times. The following figure summarizes the Continue on Error process:



The following procedures describe the use of this feature:

- [Running Continue on Error During Compilation, on page 821](#)
- [Analyzing Compilation Errors After Continue on Error, on page 822](#)
- [Running Continue on Error During Mapping, on page 826](#)

## Running Continue on Error During Compilation

This procedure shows you how to use the Continue on Error feature to reduce compiler iterations.

1. Enable the Continue on Error option for the design, by including the following command in an options file or by typing it in the console window:

```
option set continue_on_error 1
```

2. Compile the design.

The compiler first parses the syntax and checks for syntax errors; see [Compilation Process and CoE, on page 822](#) for details. If it encounters syntax errors, it errors out. You must fix these errors before rerunning compilation. Typical syntax errors are RTL code errors, and include typos in keywords, missing semicolons, mismatched beginning and ending brackets, or incorrect syntax.

After the syntax-checking stage, the compiler does not halt when it encounters an error, but completes a one-pass compilation and reports all the errors together. These non-syntax errors are related to synthesis, and include out-of-range access, missing function or task definitions, improper port or generics mappings, and coding from which sequential elements cannot be extracted.

3. Analyze and fix any syntax errors before proceeding with re-compiling.
4. If you do not have any syntax errors, analyze the non-syntax errors and fix them as described in [Analyzing Compilation Errors After Continue on Error, on page 822](#).

You can then synthesize the design. For errors found after mapping, you can isolate the modules by creating subprojects or by marking them as compile points and black-boxing them, and then fixing the errors. If you

want to continue synthesizing, just mark the error module as a black box and resynthesize.

## Compilation Process and CoE

The Continue on Error feature works after the first stage of the compilation process is complete, as described in the following table.

|                     |                                                                                                                                                                                                                                                                                                                                                             |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax Check/Parse  | The compiler parses all files in the project and reports the syntax errors found in each file. If errors are found, the compilation process terminates at this point and does not continue to the next stage. Any errors reported during this stage must be corrected before compilation can advance.                                                       |
| Elaboration/Sizing  | If a compilation error is encountered at any of these stages, the compiler converts the design unit containing the error to a black box. The compiler continues processing the remaining design units, automatically continuing to the next compilation stage without stopping at errors. It reports all errors found in these stages together, at the end. |
| Hardware Generation |                                                                                                                                                                                                                                                                                                                                                             |
| Optimization        |                                                                                                                                                                                                                                                                                                                                                             |

If errors are found in the latter three stages, these errors must be corrected before any subsequent synthesis operations can be performed.

## Analyzing Compilation Errors After Continue on Error

This procedure describes techniques to analyze errors identified after using Continue on Error at the compilation stage, as described in [Running Continue on Error During Compilation, on page 821](#). These errors are non-syntactic errors; the syntax errors must be fixed before compiling with Continue on Error.

1. Check the log file for error messages.

```

E:CG169 : CG169_svlog_lgen_0.v(6) | Fork with a disable is not synthesizable
BN:CG361 : CL224_svlog_lgen_1.v(1) | Synthesizing module CL224_svlog_1

SW:CG361 : CL224_svlog_lgen_1.v(9) | Case tag overlaps with a previous tag (case branches 0 and 1)
BN:CG364 : CG413_svlog_lgen_2.v(1) | Synthesizing module CG413_svlog_2

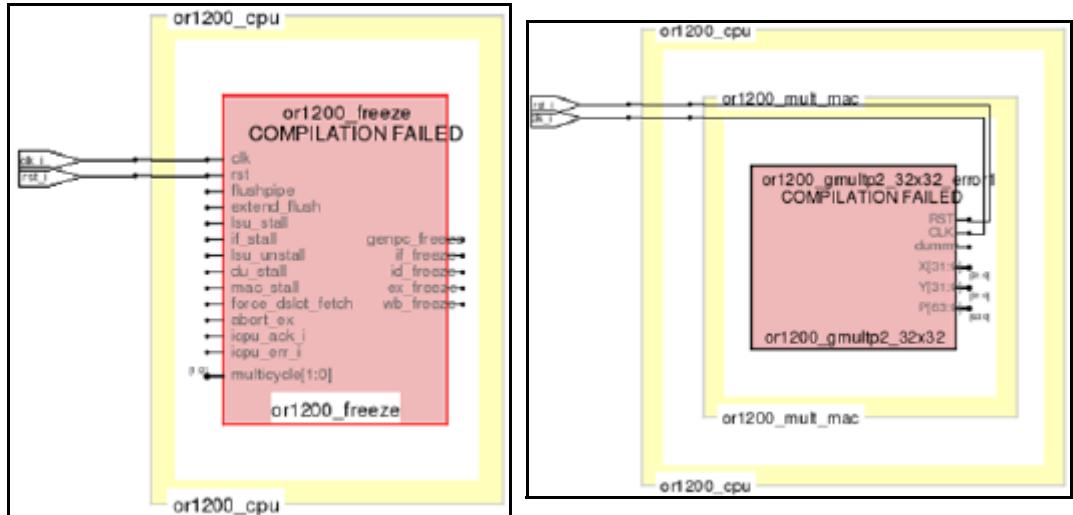
E:CG413 : CG413_svlog_lgen_2.v(13) | Assignment expressions are valid within procedural statements
BN:CG364 : top.v(1) | Synthesizing module top

BN:CG794 : top.v(6) | Using module pass_and_vhdl2008_0 from library work
BN:CG794 : top.v(7) | Using module pass_nand_vhdl2008_1 from library work
BN:CG794 : top.v(79) | Using module CD729_vhdl2008_0 from library work
BN:CG794 : top.v(84) | Using module CD293_vhdl2008_1 from library work
BN:CG794 : top.v(90) | Using module CD300_vhdl2008_2 from library work
E:CL224 : CL224_svlog_lgen_1.v(7) | ROM clash detected: address 0000000 with multiple data entries 1

At c_ver Exit (Real Time elapsed 0h:00m:01s; CPU Time elapsed 0h:00m:00s; Memory used current: 100MB)

```

- If you get a message about missing modules, include the modules before rerunning compilation. You can define modules as black boxes with the Set as Black Box command in the Design Hierarchy view. Check all modules that were black-boxed by CoE. You can use the following techniques to identify the modules in the RTL view:
  - Check all red modules in the RTL view; the color indicates modules with errors. If the module retains its original module name, the error is fully contained in the module. If the module name has an \_error suffix, the error has to do with port mismatches with the parent module. In the latter case, the parent module is black-boxed by CoE.



- Use the Tcl `find` command to search for modules with the `is_error_blackbox` property. The tool attaches this property to all modules with errors. For example, these commands list all modules with errors, and instances with errors, respectively:

```
get_prop -prop inst_of [find -hier -inst * -filter @is_error_blackbox==1]
c_list [find -hier -inst * -filter @is_error_blackbox==1]
```

3. After you have identified the errors, either fix them or isolate them.

You must deal with all CoE-identified compiler errors before proceeding with any subsequent synthesis operations. If you cannot immediately fix the errors, use these techniques to isolate and fix modules:

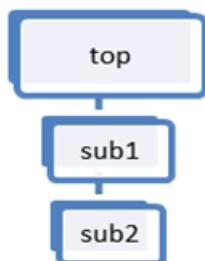
- Export the affected module as a subproject and fix it later.
- Mark it as a compile point, and define it as a black box with the Set as Black Box command in the Design Hierarchy view.

4. Re-compile the design and proceed with synthesis, using one of these approaches:

- Synthesize only the error-free modules and then merge them into the final netlist.
- Synthesize the entire design with black boxes for the modules with errors to be resolved.

## Example of Hierarchical Error Reporting

This Verilog example has two sub-modules, as shown below:



The code highlighted in red indicates the locations of the errors after compilation with CoE:

```
module sub2 (input d,rst,set,clk,
 output reg q,
 input [7:0]in, input [2:0] raddr, waddr,
 output [7:0] out3,
 input [3:0] a, b,
 output reg [4:0] q1);
 nomodule i0(in, clk);

 always@(posedge clk)
 begin
 if(clk)
 q = d << -3;
 end
 reg [7:0] mem [7:0];
 initial
 begin
 $readmemb ("data.dat",in);
 end

 always @(posedge clk)
 mem[waddr] <= in;
 assign out3 = mem[raddr];
 wire [2:0] temp;

 always@(posedge clk)
 begin
 q1 <= temp | a;
 end
 assign temp = a ** b;
 endmodule

 module sub1 (input d,rst,set,clk,
 output reg q,
 input [7:0]in, input [2:0] raddr, waddr,
 output [7:0] out3,
 input [3:0] a, b,
 output reg [4:0] q1);

 always @(posedge clk)
 begin
 q <= data1 | test; // 2 errors reported
 end
 reg q;
```

```
always @ (posedge clk)
begin
 if (rst)
 q <= 1'b0;
 else
 q = d;
 end
endmodule

module top (input d,rst,set,clk,
 output reg q,
 input [7:0]in, input [2:0] raddr, addr,
 output [7:0] out3,
 input [3:0] a, b,
 output reg [4:0] q1);
sub1 u1 (.*);
endmodule
```

## Running Continue on Error During Mapping

By default the tool runs continue on error during the mapping stage of synthesis. With this setting, the tool does not stop when it encounters an error, but maps the rest of the design.

1. To disable CoE from running during mapping, use the following command:

```
option set continue_on_error 0
```

# Including UPF Specifications

The UPF functionality described here facilitates the porting of ASIC power schemes to FPGA. Prototyping low power intent enables the functional validation of isolation and retention strategies, and power management schemes.

In this solution, power-aware FPGA synthesis is based on virtual power domains and, by its nature, does not support commands or options that control power/ground nets and power switches (see [Differences Between ASIC and FPGA Power Domains, on page 827](#)). The supported UPF commands are a subset of the IEEE 1801-2009 UPF standard, and are related to isolation, retention, power domains, and scope resolution.

The following sections describe the details:

- [Incorporating UPF in the Standard Compiler Flow](#), on page 828
- [Specifying UPF Power Domains](#), on page 830
- [Specifying Isolation Cell Strategies](#), on page 835
- [Modeling Retention Logic](#), on page 847
- [Checking the UPF Implementation](#), on page 857
- [Verifying Designs with UPF Information](#), on page 860

## Differences Between ASIC and FPGA Power Domains

FPGA power domains (PDs) differ from ASIC power domains.

| ASIC                                                          | FPGA                                     |
|---------------------------------------------------------------|------------------------------------------|
| Multiple voltages: VDD, VDD_SUB1, VDD_SUB2.                   | Single voltage: VDD.                     |
| Level shifters for signals between different voltage domains. | No level shifters.                       |
| Power switches for turning off a PD.                          | No power switches. Cannot turn off a PD. |

## UPF Limitations

The UPF functionality has some known limitations:

- UPF implementations must be applied as follows:

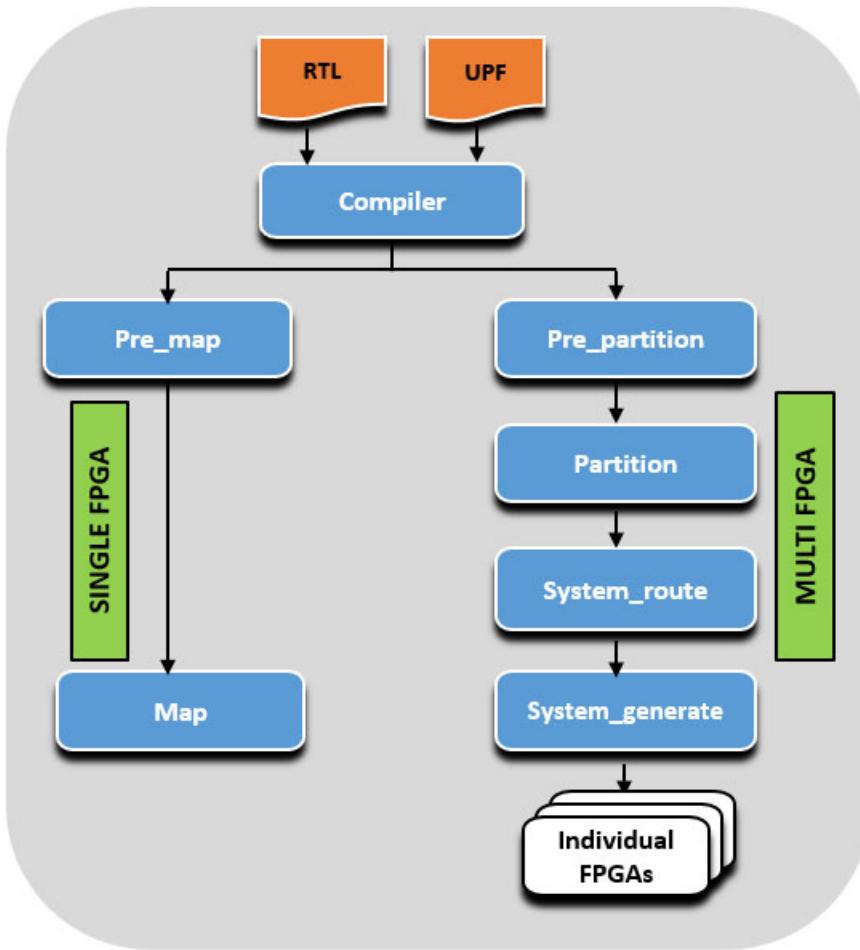
- Names of instances, registers, and ports specified for UPF must match the names generated by the compiler. Isolation strategies ignore the following conditions:
  - Bi-directional ports specified for isolation.
- Retention strategies ignore the following conditions:
  - Instantiated primitives specified for retention.
- Power domain corruption does not support instantiated primitives and the following RAM primitives inferred by the compiler:
  - True dual-port RAM (NRAM)
  - RAM1 with reset (RAM2)

## Incorporating UPF in the Standard Compiler Flow

You cannot specify the UPF commands to define ASIC low power schemes from an interactive Tcl shell; you must put the commands in a UPF file and then add the UPF file to the design.

### The UPF Design Flow

You can directly import and use UPF files to compile the prototype design, as shown below:



### Procedure to Incorporate UPF in the Native Compiler Flow

1. Create a file with the UPF-related commands, called `fileName.upf`, which models the power domains (PDs), isolation between PDs, and the retention circuitry.
  - You must specify the UPF commands in a file. This file can be the ASIC UPF file, like a UPF file from the Synopsys Design Compiler® tool. The file must be in plain text and cannot be encrypted. UPF defined for logic mapped to instantiated FPGA components, and logic for NGC cores is not supported.

- For details about supported UPF commands and their syntax, see [UPF Commands, on page 259](#) of the *Command Reference Manual*.
- Specify power domains and isolation and retention strategies, as described in [Specifying UPF Power Domains, on page 830](#), [Specifying Isolation Cell Strategies, on page 835](#), and [Modeling Retention Logic, on page 847](#).
2. Add the UPF file by specifying it as an argument to the appropriate run command, at the stages shown in the UPF flow diagram.
- You can specify the UPF file at the compile stage with the corresponding run commands. For example:
- ```
run compile -srclist add_files.sfl -upf upfFilename.upf
```
- The tool reads in the UPF information at the design stage specified. The netlist generated after mapping includes the UPF information.
- The tool does not optimize the retention and isolation logic generated because of constant propagation. You cannot configure retention and isolation cell mapping on FPGAs as you can with SoC designs. If you have ASIC map_isolation_cell commands, they are ignored.
- 3. Check that the UPF information was correctly processed by examining the UPF report file and the schematic, as described in [Checking the UPF Implementation, on page 857](#).
 - 4. Verify the UPF implementation by running simulation. See [Verifying Designs with UPF Information, on page 860](#) for details.

Specifying UPF Power Domains

Power domains are required to identify isolation ports and power domain elements for isolation strategies. You can specify power domains from the top down or from the bottom up, as shown in the following examples. The level from which you define the domain determines the scope. The default scope is the top level. For examples, see the following:

- [Specifying Power Domains from the Top Down, on page 831](#)
- [Specifying Power Domains from the Bottom Up, on page 832](#)
- [Specifying Power Domains with Multiple Elements, on page 833](#)
- [Setting the Scope for create_power_domain, on page 833](#)

Specifying Power Domains from the Top Down

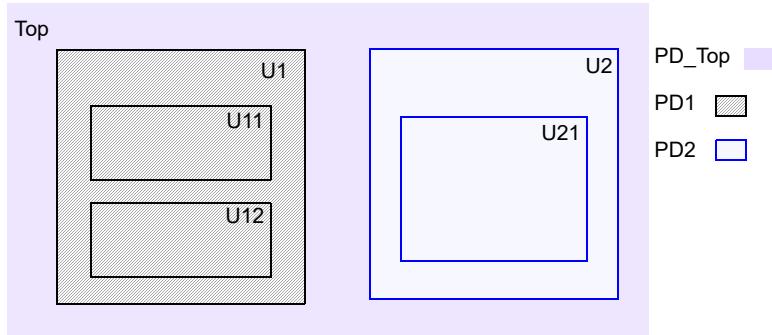
To specify power domains from the top down, use the `create_power_domain` command. For the syntax of the UPF commands mentioned in the examples, see [UPF Commands, on page 259](#) of the *Command Reference Manual*.

The commands shown below use the top-down methodology and define a power domain called PD_Top, which includes two internal domains (PD1 and PD2). Power domains include all modules contained within them, unless explicitly specified otherwise with the `-elements` option. The scope is assumed to start with the current level, unless explicitly specified otherwise with the `-scope` option.

Example 1

The example starts from Top, which is the default scope for the commands. The commands create one power domain (PD1) with U1, and another power domain (PD2) with U2. You create one top-level upf file for the design.

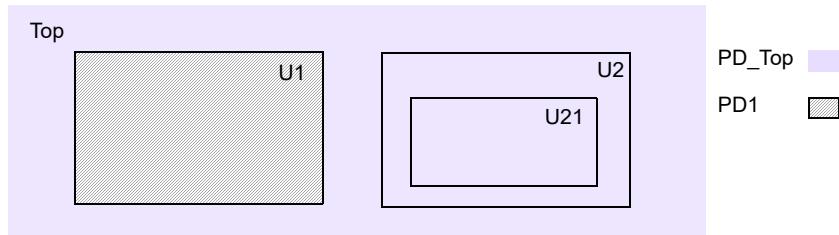
```
# Current scope is Top
create_power_domain PD_Top -include_scope
create_power_domain PD1 -elements {U1}
create_power_domain PD2 -elements {U2}
```



Example 2

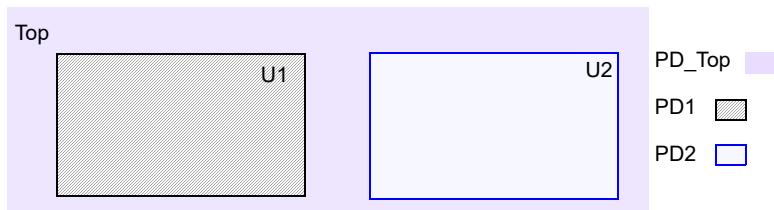
In the following example, the U1 module has its own power domain (PD1), but the U2 module is included in the top-level power domain. The `set_scope` command explicitly switches the scope to specify the power domain, instead of using the `-elements` option as in the previous example. See [Setting the Scope for create_power_domain, on page 833](#) for more information about scope.

```
# Current scope is Top  
create_power_domain PD_Top -include_scope  
# Explicitly set the scope to U1  
set_scope U1  
# Current scope is U1  
create_power_domain PD1
```



Specifying Power Domains from the Bottom Up

To specify power domains from the bottom up, define power domains for each separate module (U1 and U2) with `create_power_domain` commands. There will be one top-level upf file that sources the module-level UPF files, U1.upf and U2.upf, respectively.



Define a power domain for the top level, and load the module-level UPF files. The sample files below illustrate the details.

U1.upf File

```
create_power_domain PD1
```

U2.upf File

```
create_power_domain PD2
```

Top.upf File

```
# Load lower level upf to current scope, which is top
load_upf U1.upf -scope U1
load_upf U2.upf -scope U2
# Define top-level power domain
create_power_domain PD
```

Specifying Power Domains with Multiple Elements

To create a power domain that consists of a particular collection of cells, use the `-elements` argument. You can specify a single element or multiple elements. If you specify multiple elements, separate them with spaces. The following command creates a power domain called PD1 that includes the cells in U1 and U2:

```
create_power_domain PD1 -elements {U1 U2}
```

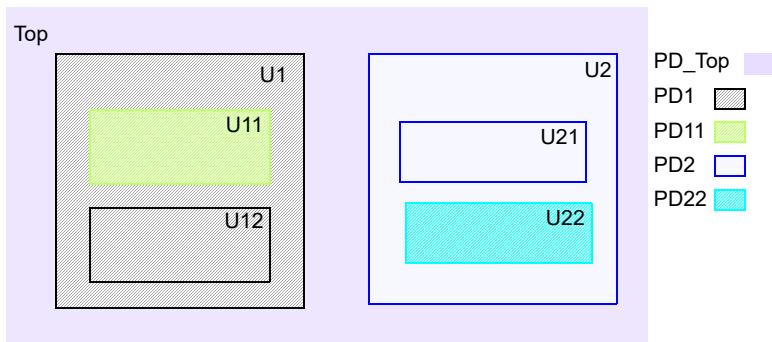
Defining the Power Domain Supply

The supply for a power domain can be defined as individual nets or bundles of nets called supply sets. Identify the supply for the power domain using the commands listed below. For descriptions of the commands and their syntax, refer to [Chapter 5, UPF Commands](#) in the *Command Reference Manual*.

| To ... | Use This Command ... |
|--|--|
| Define a supply net | <code>set_domain_supply_net</code> |
| Define a supply set | <code>create_supply_set</code> <code>create_power_domain -supply</code> (See Defining Supply Sets , on page 843.) |
| Associate a supply set with a power domain | <code>set_domain_supply_net</code> <code>associate_supply_set</code> |

Setting the Scope for `create_power_domain`

Either set the scope for power domain creation explicitly with the `set_scope` command, or implicitly, by specifying a path from the current scope. Both the examples below create the power domains shown in the following figure, but they use different methods. You can use hierarchical names.



Explicit Scope

```
# Current scope is top
create_power_domain PD_Top -include_scope
# Change scope to U1
set_scope U1
# Designate U1 a power domain
create_power_domain PD1
# Change scope and designate U11 as a power domain
set_scope U11
create_power_domain PD11
# Change scope and designate U2 as a power domain
set_scope ../../U2
create_power_domain PD2
# Change scope and designate U22 as a power domain
set_scope U22
create_power_domain PD22
# Set scope to top
set_scope
```

Implicit Scope

```
# Designate U1 as a power domain from current scope, which is top
create_power_domain PD_Top -include_scope
create_power_domain PD1 -scope {U1}
# Specify U11 as power domain, with elements specified with path
# from current scope
create_power_domain PD11 -scope {U1} -elements {U1/U11}
# Specify U2 as a power domain. Current scope is still top.
```

```
create_power_domain PD2 -scope {U2}
# Specify U22 as power domain, with elements specified with path
from current scope
create_power_domain PD22 -scope {U2} -elements {U2/U22}
# Current scope is still top
```

Specifying Isolation Cell Strategies

Unexpected values can result when the output of powered-off modules drives active, powered-on modules, and this can cause incorrect logic behavior. Isolation is a strategy used to separate design elements with power from parts of the design that are still active and have power. The UPF standard allows for the insertion of isolation cells between power domains. The cells are powered by a constant supply, and “clamp” the output of the powered-off domain to ensure a fixed value during its powered-down state. You must specify at least one isolation strategy for every power domain created.

See the following for details:

- [Specifying Isolation Cells](#), on page 835
- [Avoiding Redundant Isolation Logic](#), on page 840
- [Defining Supply Sets](#), on page 843
- [Forcing Corruption in Selected Power Domains](#), on page 845
- [Setting Corruption for Specific Power Domains](#), on page 881

Specifying Isolation Cells

The following procedure provides some guidelines for specifying isolation cells; for the syntax to the commands, see [Chapter 5, UPF Commands](#) of the *Command Reference Manual*.

1. Define power domain with the `create_power_domain` command, as described in [Specifying UPF Power Domains](#), on page 830.
2. Specify at least one isolation strategy for every non-default power domain.

See [UPF Limitations](#), on page 827 for some limitations.

3. To apply isolation strategies to supply sets instead of individual supply nets, define the supply sets (bundles of supply nets).

See [Defining Supply Sets, on page 843](#).

4. Define equivalent supply nets or supply sets with the `set_equivalent` command.
 - Use `set_equivalent -nets` command to declare equivalent nets, and `set_equivalent -sets` to declare equivalent sets.

You can use the equivalency declarations with different isolation strategies; for examples, see the command description in [Chapter 5, UPF Commands](#) of the *Command Reference Manual*.
 - To specify equivalence across scopes, specify the command as shown in the following example:

```
create_supply_net SN1
set_scope u1
create_supply set SS2
create_supply net SN2
set_scope /
# Make set SS1 equivalent to SS2 in another scope:
set_equivalent -sets {SS1 u1/SS2}
# Make net SN1 equivalent to SN2 in another scope:
set_equivalent -nets {SN1 u1/SN2}
```

 - To specify multiple `set_equivalent` commands for setting equivalent elements, define them as shown below:

```
create_supply_set SS1
create_supply_set SS2
create_supply_set SS3
create_supply_set SS4
# Make sets SS1, SS2, and SS3 equivalent:
set_equivalent_sets {SS1 SS2 SS3}
# Make set SS1 equivalent to SS4, thus making all the sets equivalent:
set_equivalent_sets {SS1 SS4}
```
5. Set a global isolation strategy and use local commands to override the global strategy on specific modules.

The following example inserts isolation logic on all the output ports of the PD1 domain, except for port1 and port2.

```
set_isolation PD1_iso -domain PD1 -clamp_value 1 -applies_to
outputs
set_isolation PD1_no_iso -domain PD1 -no_isolation -elements
{port1 port2}
```

6. Use the `set_isolation` and `set_isolation_control` commands in combination to specify isolation cells.
 - Apply the commands to design ports or pins. You cannot specify isolation on bidirectional ports. If multiple commands apply for the same component, the tool processes the first one and ignores the rest.
 - To avoid the insertion of redundant isolation logic, use the techniques described in [Avoiding Redundant Isolation Logic, on page 840](#).
 - Use the `-location` option of the `set_isolation_control` command to specify the location of the isolation cell. See [Isolation Cell Location Examples, on page 839](#) for some examples.
7. Set other isolation options with the `set_isolation` command.
 - Specify the value to be used during the powered-down state in the `-clamp_value` option of the `set_isolation` command. You can set it to 0, 1, or latch. See [set_isolation, on page 279](#) in the *Command Reference Manual* for examples of the results with different settings.
 - To apply isolation logic to all the outputs on a power domain boundary, specify the `set_isolation` command as shown below:


```
set_isolation PD1_iso -domain PD1 -clamp_value 1
-applies_to outputs
```
 - To insert isolation logic on specified ports and override `-applies_to`, specify the command with the `-elements` argument, as these example show:


```
set_isolation PD1_iso -domain PD1 -clamp_value 1
-applies_to outputs -elements {U1/port1 U1/port2}
```
 - To specify that no isolation logic be inserted on certain ports, use command syntax like this:


```
set_isolation PD1_iso -domain PD1 -applies_to outputs
-elements {U1/port1 U1/port2} -no_isolation
```

In this case, no isolation logic is inserted on U1/port1 and U2/port2, and `-applies_to` is ignored.

8. Use the `set_isolation` command in conjunction with the following Tcl variables to control how the `-applies_to` option is implemented for the isolation strategy of a power domain. You can use the Tcl variables as follows:

- `enable_upf_1_0_applies_to_default_output` when the `-applies_to` option is not specified. For example, if you specify the following Tcl command:

```
option set -enable_upf_1_0_applies_to_default_output false
```

With the isolation strategy below, then both the input and output ports are instrumented for the power domain.

```
set_isolation ISO -domain PD -clamp_value 1
set_isolation_control ISO -domain PD -isolation_signal
    iso_ctrl -location parent
```

- `upf_iso_filter_elements_with_applies_to` when the `-applies_to` option is specified with the `-elements` option for the power domain. For example, if you specify the following Tcl command:

```
option set -upf_iso_filter_elements_with_applies_to error
```

With the isolation strategy below, this generates an error message.

```
set_isolation ISO -domain PD -elements
    {pin1 pin2 pout1 pout2} -applies_to outputs
    -clamp_value 1
set_isolation_control ISO -domain PD
    -isolation_signal iso_ctrl -location parent
```

For this example, if you specify the following Tcl command:

```
option set -upf_iso_filter_elements_with_applies_to disable
```

Then the isolation strategy below ignores the `-applies_to` option and all elements `{pin1 pin2 pout1 pout2}` are instrumented for the power domain.

```
set_isolation ISO -domain PD -elements
    {pin1 pin2 pout1 pout2} -applies_to outputs -clamp_value 1
set_isolation_control ISO -domain PD
    -isolation_signal iso_ctrl -location parent
```

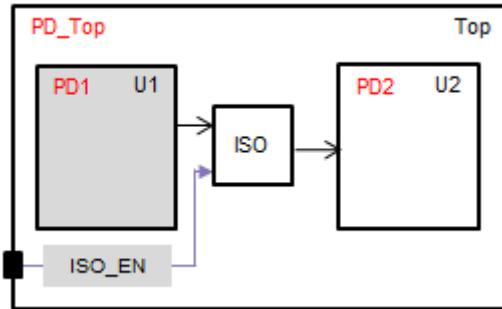
For a complete description of how these Tcl variables can be used to implement the isolation strategy for a domain, see [option, on page 81](#) in the *ProtoCompiler Command Reference*.

Isolation Cell Location Examples

Different values for the `set_isolation -location` command produce different results.

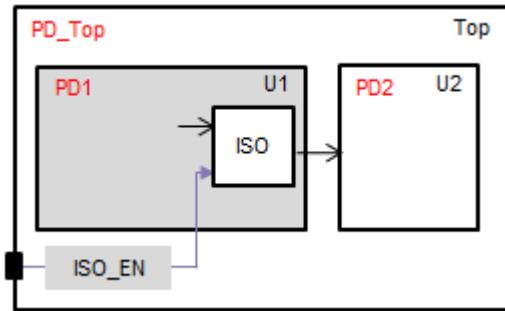
- `set_isolation -location parent:`

```
set_isolation ISO_on_outputs -domain PD1 -clamp_value 1  
-applies_to outputs  
  
set_isolation_control ISO_on_outputs -domain PD1  
-isolation_signal iso_en -isolation_sense high -location  
parent
```



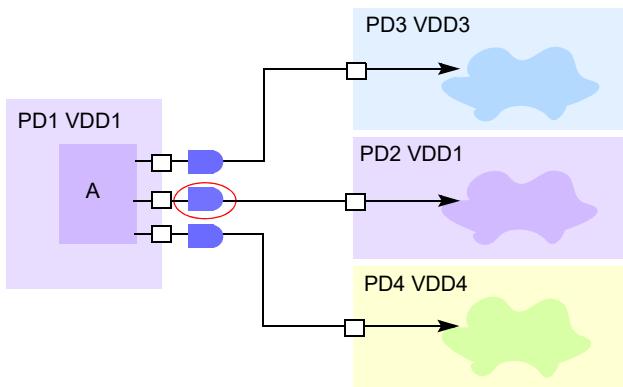
- `set_isolation -location self:`

```
set_isolation ISO_on_outputs -domain PD1 -clamp_value 1  
-applies_to outputs  
  
set_isolation_control ISO_on_outputs -domain PD1  
-isolation_signal -iso_en -isolation_sense high  
-location self
```



Avoiding Redundant Isolation Logic

By default, the tool inserts isolation logic on all output ports of power domain PD1. The figure shows one scenario where isolation logic is not needed. PD2 is powered by the same supply (VDD1) as PD1, and does not need the isolation logic that would be inserted by default (circled in red).



The flexible isolation and other isolation techniques described below show you how to handle redundant isolation. Some techniques define isolation strategies for supply sets (bundles of supply nets) instead of individual supply nets. See [Chapter 5, UPF Commands](#) of the *Command Reference Manual* for the complete syntax of all commands mentioned.

1. To add isolation logic only if the driver and receiver supply sets are different, do the following:
 - Define supply sets, as described in [Defining Supply Sets, on page 843](#).

- Specify the `set_isolation` command with the `-diff_supply_only true` argument.
- Specify the `set_isolation_control` command as usual. For example:

```
set_isolation -applies_to outputs -diff_supply_only true
set_isolation_control -location parent
```

For more examples, see [diff_supply_only Examples, on page 841](#).

2. To specify the insertion of isolation logic between a source and a sink, do the following:

- Define supply sets, as described in [Defining Supply Sets, on page 843](#).
- Specify the `set_isolation` command with the `-source` and `-sink` arguments.
- Specify the `set_isolation_control` command as usual. For example:

```
set_isolation -applies_to outputs -source SS_VDD1 -sink SS_VDD3
set_isolation_control -location parent
```

- If the `set_isolation_control -location self | parent` command prevents the tool from isolating the specified path, use the `-location fanout` option instead:

```
set_isolation -applies_to outputs -sink SS_VDD2
set_isolation_control -location fanout
```

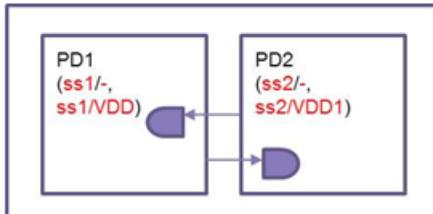
3. To remove redundant logic on a supply net, use the `set_isolation -no_isolation` command:

```
set_isolation PD1_iso -domain PD1 -applies_to outputs
-elements {U1/port1 U1/port2} -no_isolation
```

diff_supply_only Examples

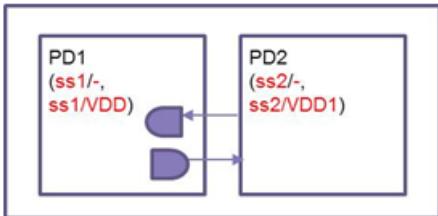
- Example 1

```
set_isolation iso_PD1 -domain PD1 -applies_to both -diff_supply_true only
set_isolation_control iso_PD1 -domain PD1 -isolation_signal iso -location fanout
```



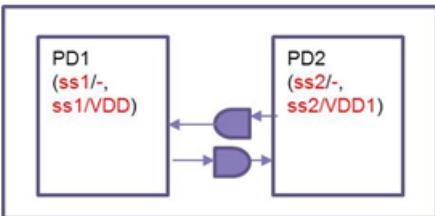
- Example 2

```
set_isolation iso_PD1 -domain PD1 -applies_to both -diff_supply_true only
set_isolation_control iso_PD1 -domain PD1 -isolation_signal iso -location self
```



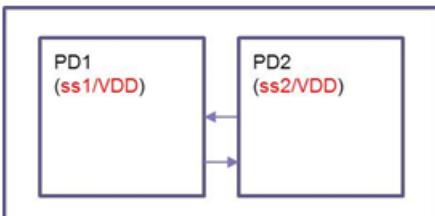
- Example 3

```
set_isolation iso_PD1 -domain PD1 -applies_to both -diff_supply_true only
set_isolation_control iso_PD1 -domain PD1 -isolation_signal iso -location parent
```



- Example 4

```
set_isolation iso_PD1 -domain PD1 -applies_to both -diff_supply_true only
set_isolation_control iso_PD1 -domain PD1 -isolation_signal iso -location
self/parent/fanout
```



Defining Supply Sets

The UPF 2.0 standard allows for the insertion of isolation cells between power domains based on the supply set for a power domain, not just a supply net. A supply set is a bundle of supply nets. The software supports two predefined supply set functions: power and ground. Each function can correspond to only one supply net.



You can define supply sets either explicitly (`create_supply_set`) or implicitly with set handles (`create_power_domain`). For the complete syntax of all commands mentioned, refer to [Chapter 5, UPF Commands](#) in the *Command Reference Manual*.

1. Use the `create_supply_set` to explicitly define supply sets.

```
create_supply_set myss
```

- Specify a simple name for the supply set. Do not use names qualified with a hierarchy separator, like a slash (/) or a period (.). Supply set names like `string1.string2` or `string1/string2` cause an error message to be generated.
- For each explicit supply set defined, make sure to also specify the corresponding power and/or ground functionality with the `create_supply_set -function` command. For example:

```
create_supply_set myss -function {power VDD} -function {ground GND}
```

- Optionally, use the `-update` argument to update an existing supply set definition with the supply net information.
- Associate the supply set with the power domain using the `set_domain_supply_net` or `associate_supply_net` commands. For details, see the examples in [For more information, see the supply set assignment examples that follow. Supply Set Assignment: Example 1, on page 844](#) and [Supply Set Assignment: Example 2, on page 845](#).

2. To implicitly define supply sets, do the following:

- Define the supply set with `create_supply_set`, as described above.

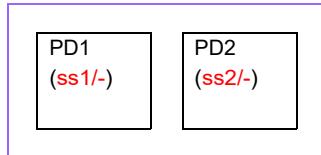
- Specify the `create_power_domain -supply` command and the set handle to automatically create the supply set and associate it with the power domain. When a power domain is defined, the tool automatically creates these supply set handles: `domain.primary`, `domain.default_isolation`, and `domain.default_retention`.

This is an example of the command with a supply set handle:

```
create_power_domain PD1 -supply {primary ss1}
```

For more information, see the supply set assignment examples that follow.
Supply Set Assignment: Example 1

You can use different command combinations to define supply sets (ss) for power domains (PD) and achieve the results in the figure below:



- `create_supply_net` and `set_domain_supply_net`

```
create_supply_set ss1
create_supply_set ss2
set_domain_supply_net PD1 -primary_power_net ss1.power
    -primary_ground_net ss1.ground
set_domain_supply_net PD2 -primary_power_net ss2.power
    -primary_ground_net ss2.ground
```

- `create_supply_net` and `associate_supply_net`

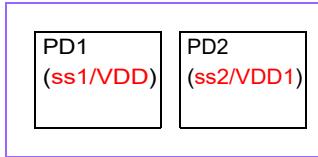
```
create_power_domain PD1
create_power_domain PD2
create_supply_set ss1
create_supply_set ss2
associate_supply_set ss1 -handle PD1.primary
associate_supply_set ss2 -handle PD2.primary
```

- `create_supply_net` and supply set handles

```
create_supply_set ss1
create_supply_set ss2
create_power_domain PD1 -supply {primary ss1}
create_power_domain PD2 -supply {primary ss2}
```

Supply Set Assignment: Example 2

You can use different command combinations to define supply sets (ss) for power domains (PD) to achieve the results below:



- create_supply_net and set_domain_supply_net

```
create_supply_set ss1 -function {power VDD} -function {ground GND}
create_supply_set ss2 -function {power VDD1} -function {ground GND}
set_domain_supply_net PD1 -primary_power_net ss1.power
    -primary_ground_net ss1.ground
set_domain_supply_net PD2 -primary_power_net ss2.power
    -primary_ground_net ss2.ground
```

- create_supply_net and supply set handles

```
create_supply_set ss1 -function {power VDD} -function {ground GND}
create_supply_set ss2 -function {power VDD1} -function {ground GND}
create_power_domain PD1 -supply {primary ss1}
create_power_domain PD2 -supply {primary ss2}
```

Use the -update option only if the supply set is being updated and not when it is first used.

Forcing Corruption in Selected Power Domains

When ASIC power domains are powered off, the domain state becomes unknown or corrupted, but the isolation and retention strategies restore the state when it is powered on again. Forced power domain corruption is a technique used to increase test coverage of the FPGA design and add confidence that the design will continue to function reliably when the ASIC is fabricated. You can implement logic to force the sequential elements to an unknown state when the domain is shut off by following these steps:

1. Add the `create_power_switch` command to a power domain.

See [create_power_switch, on page 267](#) of the *Prototyping Command Reference* for syntax details. If you do not specify this command, corruption is not enabled.

Corruption is applied globally, but you can disable specific power domains or instances from corruption using the following steps.

2. For the native compiler flow, follow these steps to selectively enable corruption for power domains and instances.
 - Create an additional UPF script file (e.g. `corrupt_upf`).
 - In this file, specify the `corrupt_pd` command ([corrupt_pd, on page 262](#) of the *Prototyping Command Reference*) with the power domain or instance you want to enable.

```
corrupt_pd -domain myRAM
```

- You can also enable random corruption of sequential elements by setting the mode. In this example, random corruption is applied to power domain PD3:

```
corrupt_pd -domain PD3 -mode random
```

- Run compile with the top-level design UPF file and `corrupt.upf` file. Source the `corrupt.upf` file after the top-level design UPF file.

```
run compile -srclist add_files.sfl -top_module top  
-upf design_top.upf -upf corrupt.upf run pre_map -fdc  
design.fdc -upf design_top.upf -upf  
corrupt.upf
```

```
run pre_partition -fdc design.fdc -upf design_top.upf -upf  
corrupt.upf
```

3. For the UC flow, follow these steps to enable corruption:

- Specify the appropriate `-power` options at the command line in the VCS script when you specify the UPF file. Use `[-power=seqcorrupt]` for register logic and `[-power=hw_corrupt_memory]` for memory corruption.
- Set the type of corruption for both logic and memory by setting this attribute:

```
set_design_attributes -attribute SNPS_random_corruption 0|1|I
```

The default is 0, which implements all zeros corruption. A value of 1 implements all ones corruption, and I implements inversion-based corruption. The implementation happens when the design is compiled.

- For boundary mux corruption, specify this -power option at the VCS command line: [-power=enable_boundary_gates].

Modeling Retention Logic

Components can be switched on and off to reduce power consumption. When power is shut off, the state value is lost. Some components require that these values be retained, to avoid a performance hit or to ensure quick restarts when they are powered on again. The UPF standard takes this into account by allowing you to specify flip-flops to retain the state value.

To define a retention strategy, you must specify a combination of commands: the `set_retention` command, and either the `set_retention_control` or `map_retention_cell` commands, depending on the strategy you select.

See the following for details:

- [Steps for Modeling Retention Logic](#), on page 847
- [Using the Default Retention Scheme](#), on page 849
- [Using a User-Defined Retention Scheme \(`map_retention_cell`\)](#), on page 854
- [UPF Commands](#), on page 259 of the *Command Reference Manual* (command syntax descriptions)
- [UPF Limitations](#), on page 827

Steps for Modeling Retention Logic

To define a retention strategy, you must specify a combination of commands: the `set_retention` and `set_retention_control` commands, and optionally the `map_retention_cell` command, according to the strategy you select.

1. For each retention strategy, specify the `set_retention` command.
 - Define the power domain with the `set_retention` command. A power domain can contain either a single retention strategy, or multiple

strategies. Use a command like the one shown below to define the power domain:

```
set_retention ret_pd1 -domain pd1.
```

- Apply the commands to sequential elements like flip-flops, RAMs, state machines, or shift registers. The tool models the retention logic according to the various sequential elements in the design.
- To convert all registers under a particular power domain, do not use the `-elements` option of the `set_retention` command. If you specify `-elements`, the tool converts registers under the instances in the element list.

```
set_retention PD1_ret -domain PD1
```

- To specify registers you do not want to convert, use the `-no_retention` option of the `set_retention` command.
- 2. You can specify retention on individual bits, using syntax like the syntax for bus elements [5:0] shown here:

```
set_retention ret1_pd1 -domain PD1 -elements {count[0]}  
set_retention ret2_pd1 -domain PD1 -elements {count[4:3]}  
set_retention ret3_pd1 -domain PD1 -elements {count[2:1]}  
set_retention ret4_pd1 -domain PD1 -elements {count[5]}
```

- 3. Use local commands on individual modules to override the global setting as needed.

The following example converts all registers under PD1 except for R1, R2, and R3:

```
set_retention PD1_ret -domain PD1
```

```
set_retention PD1_no_ret -domain PD1 -no_retention -elements {R1 R2 R3}
```

- 4. For each power domain, select a retention strategy using the appropriate commands.

A power domain can contain a single retention strategy or multiple strategies. You can also use a mixture of the two retention schemes.

- Single-signal, default-based retention scheme (`set_retention_control`)
Single-signal schemes have the same save and restore signals. See [Using the Default Retention Scheme, on page 849](#) for details.

- User-defined retention scheme (`set_retention_control` and `map_retention_cell`)

You can use this method for dual-signal retention schemes, which have different save and restore signals. See [Using a User-Defined Retention Scheme \(`map_retention_cell`\), on page 854](#) for information on setting this up.

If multiple commands apply to the same component, the tool processes the first command, issues a warning message, and ignores the subsequent commands.

Using the Default Retention Scheme

This retention scheme uses default values and can only be used to specify single-signal retention (same save and restore signals). For dual-signal retention (different save and restore signals), use the `map_retention_cell` command ([Using a User-Defined Retention Scheme \(`map_retention_cell`\), on page 854](#)).

1. Specify the power domain with the `set_retention` command.

See [Steps for Modeling Retention Logic, on page 847](#) for details.

2. Specify different senses for the save and restore signals with the `set_retention_control` command.

```
set_retention_control PD1_ret -domain PD1 -save_signal {signal_ret high}  
-restore_signal {signal_ret low}
```

This scheme does not support dual-signal retention.

3. Specify objects for retention with the `set_retention_control` command, applying it to sequential elements like flip-flops, or to RAMs, ROMs, state machines, and shift registers.

```
set_retention ret_pd1 -domain pd1  
set_retention_control ret_pd1 -domain pd1 -save_signal {signal_ret high}  
-restore_signal {signal_ret low}
```

The following table describes retention polarity with `set_retention_control`:

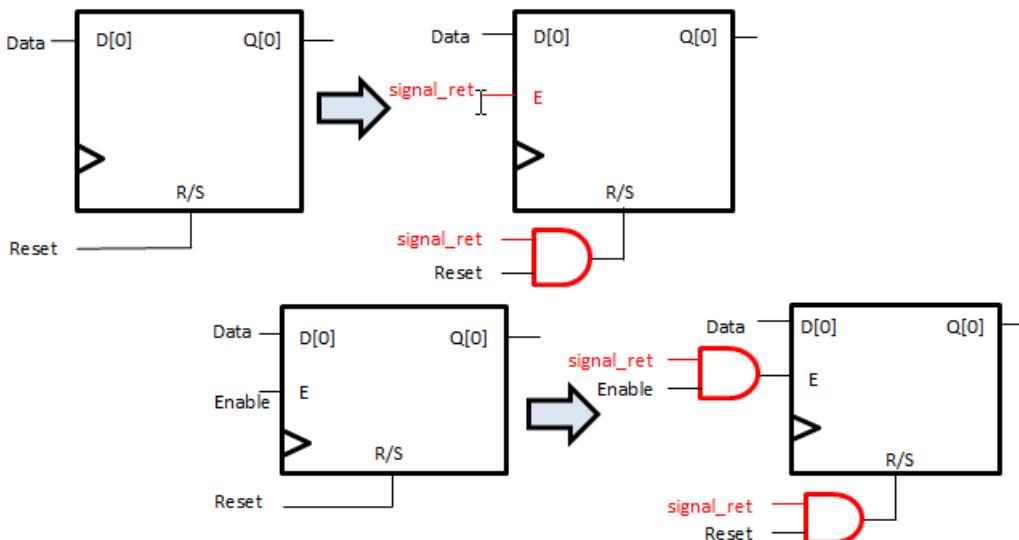
| signal_ret | clk | Q |
|-------------------|------------|-----------|
| 1 | Active | Follows D |
| 0 | x | Holds D |

With this retention strategy, the tool uses default settings for different objects. See the examples that immediately follow this procedure for the defaults used with different objects.

`set_retention_control`: Flip-Flops

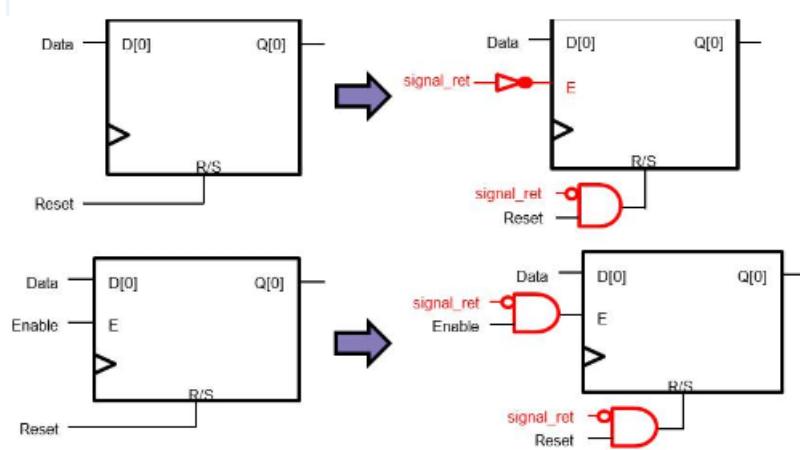
You must specify a different sense for each signal using the `set retention_control` command. The following figure shows the results when the save signal is set to high and the restore signal to low:

```
set_retention_control PD1_ret -domain PD1 -save_signal
{signal_ret high -restore_signal {signal_ret low}}
```



The next figure shows the results when the save signal is set to low and the restore signal to high:

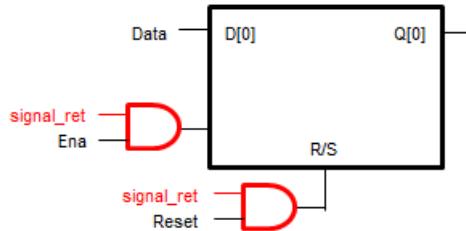
```
set_retention_control PD1_ret -domain PD1 -save_signal
{signal_ret low -restore_signal {signal_ret high}}
```



Flip-flops without enables (dff, dffr, dffs, and so on) are converted to the corresponding flip-flops with enables (dffe, dffre, dffirse, etc.).

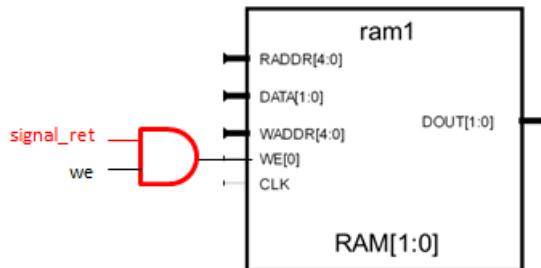
`set_retention_control: Latches`

The tool handles latches as shown below, with the state of a specified save signal (`signal_ret`) output as the restore signal.



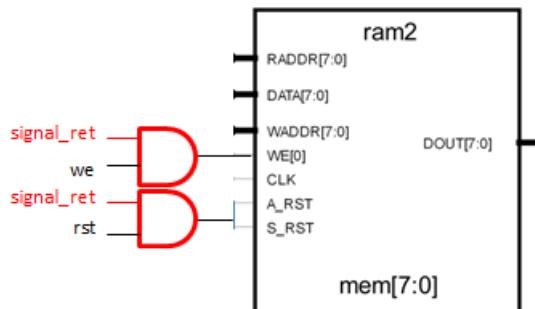
set_retention_control: RAM1

RAM1 is a structure that the compiler infers for single-port RAM. The tool handles retention for RAM1 by adding a retention signal:



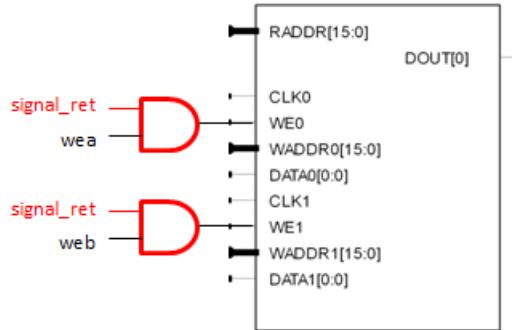
set_retention_control: RAM2

RAM2 is a structure that the compiler infers for dual-port RAM. The tool handles retention for RAM2 by adding signals:



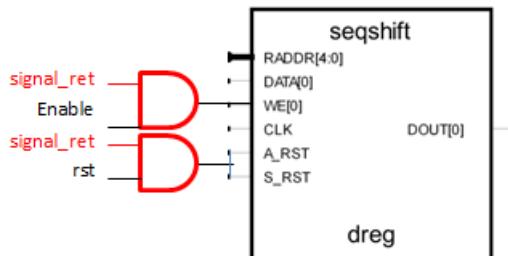
set_retention_control: NRAM

NRAM is a structure that the compiler infers for dual-port RAM where all the writes are to one process, or if there are more than two ports. The tool handles retention for NRAM by adding signals:



set_retention_control: Sequential Shifters

The compiler infers sequential shifter structures, and the tool handles retention for inferred sequential shifters by adding retention signals:



set_retention_control: State Machines

The tool decomposes state machines inferred by the compiler into registers and then applies the register retention scheme to them.

Using a User-Defined Retention Scheme (map_retention_cell)

The basis of this method is the addition of the map_retention_cell command, which lets you set up a user-defined retention scheme instead of the default-based one, or to use a dual-signal retention scheme (different save and restore signals). You must use map_retention_cell for dual-signal retention schemes.

1. Use the set_retention and set_retention_control command to set up the power domain and signals as for the default scheme.
 - Specify the power domain with the set_retention command. See [Steps for Modeling Retention Logic, on page 847](#) for details.
 - Specify different senses for the save and restore signals with the set_retention_control command. This is the same as for the default scheme. See [Using the Default Retention Scheme, on page 849](#) for details.
2. Define RTL retention models for primitives and add them to the design.
 - Define the retention models in an RTL source file.
 - Specify retention models for the sequential elements defined in the RTL source file through custom directives. The tool supports the following types of sequential elements: latches, flip-flops, registers, registers with pattern reset, seqshift, FSM, and RAM. You can use the following directives to define retention models for them:

| | |
|-----------------------|---|
| syn_implement | Preserves retention models in the log file after the compile stage |
| syn_ret_type | Define the type of sequential element to which the retention scheme applies |
| syn_ret_lib_cell_type | Must be set to the same value as the -lib_cell_type option of the map_retention_cell command |
| syn_upf_ret_port_type | Specify a port type. Ensure that the port names of user-defined cells match those of the elements you want to retain. |

Refer to [Attributes and Directives Summary, on page 473](#) in the *Compiler and Mapper Guide* for the complete syntax details for the directives. This is an example of how to specify the directives in the RTL:

```
module ret_dff () /* synthesis syn_implement = "1"
                     synthesis syn_upf_ret_type = "DFF" */;
  ...
endmodule

module ret_dffe () /* synthesis syn_implement = "1"
                     synthesis syn_upf_ret_type = "DFFE" */
  ...
endmodule
```

- Add the retention models to your source file list.

```
add_file -verilog "RTLsource"
add_file -verilog "retentionModel.v"
```

3. In the UPF file, specify the retention library cell that corresponds to the RTL you just created, using the `-lib_cells` option of the `map_retention_cell` command.

This step defines the equivalent FPGA retention logic for the cells. This is an example of the retention logic for the cells, specified with the `map_retention_cell -lib_cells` command:

```
map_retention_cell RET_PD -domain PD -lib_cells
  {ret_dff ret_dffr ret_dffre}
```

This is the corresponding RTL, created in the previous step with the retention directives:

```
module ret_dff () /* synthesis syn_upf_ret_type =
                     "DFF" */;
endmodule

module ret_dffr () /* synthesis syn_upf_ret_type =
                     "DFFR" */;
endmodule

module ret_dffre () /* synthesis syn_upf_ret_type =
                     "DFFRE" */;
endmodule
```

For more examples, see [Examples of Custom Retention Models, on page 856](#). If the tool does not find custom retention cells, or if the speci-

fied cells are not compatible with the RTL, the tool uses the default retention scheme.

4. To ensure that the tool uses the specified retention cells, specify this command in the top-level UPF file:

```
set use_map_retention_cell 1
```

If you do not specify this command, the tool uses the default retention models. When you specify this command, the tool uses the specified retention cells and issues a warning if it does not find the cells.

Examples of Custom Retention Models

Example 1:

```
module ret_dffrse (Q, CLK, SAVE, RESTORE, D, S, R, E)
/* synthesis syn_upf_ret_type = "DFFRSE" */;

output Q;
input CLK /* synthesis syn_upf_ret_port_type = "CLK" */;
input SAVE /* synthesis syn_upf_ret_port_type = "SAVE" */;
input RESTORE/* synthesis syn_upf_ret_port_type = "RESTORE" */;
input D ;
input S /* synthesis syn_upf_ret_port_type = "SET, ASYNC" */;
input R /* synthesis syn_upf_ret_port_type = "RESET, ASYNC" */;
input E /* synthesis syn_upf_ret_port_type = "EN" */;

logic

endmodule
```

Example 2:

```
module ret_sdffrse (Q, CLK, SAVE, RESTORE, D, S, R, E)
/* synthesis syn_upf_ret_type = "SDFFRSE" */;

output Q;
input CLK /* synthesis syn_upf_ret_port_type = "CLK" */;
input SAVE /* synthesis syn_upf_ret_port_type = "SAVE" */;
input RESTORE/* synthesis syn_upf_ret_port_type = "RESTORE" */;
input D ;
input S /* synthesis syn_upf_ret_port_type = "SET, SYNC" */;
input R /* synthesis syn_upf_ret_port_type = "RESET, SYNC" */;
```

```
input E /* synthesis syn_upf_ret_port_type = "EN" */;  
logic  
endmodule
```

Checking the UPF Implementation

The definitive test for verifying your UPF implementation is to run simulation ([Verifying Designs with UPF Information, on page 860](#)), but you can check that the UPF information was processed correctly within the prototyping tool before you run simulation. These checking techniques are described below:

1. Specify the UPF file with the appropriate design phase run command.

For details, see [Specifying UPF Power Domains, on page 830](#).

2. Specify verbose mode with this command: option set upf_verbose 1.

With this option, the tool prints a verbose message for every command parsed in the UPF files.

3. Open the UPF report, using one of these methods:

- Use the view report Tcl command.
- To open the report from the GUI, select View->Report View or click the View Reports icon.

From the `compile` report, double-click Design Power Intent Information (upf.rpt) to open the UPF report file.

4. Check the UPF report file.

- Check the details of the implementation in this report, which includes a hierarchical UPF summary, the power domains created, the isolation and retention strategies used, and elements that were not found in the design. It also provides information about the power domain supply and corruption. For a detailed description of this file and the terminology used, see [UPF Report, on page 183](#) in the *Reference Manual*.

Check that the power domains were implemented as intended.

For the isolation/retention strategies, check the Default/Explicit status to make sure the strategy was defined correctly, and the Mappable status to make sure the associated logic was implemented as intended.

```

----- UPF Report -----
----- Run Time : 0.004 -----
----- Number of power domains : 2 -----
-----
1)
Name      : PDI
Scope     : top
Mapped    : TRUE

Mapped elements : inst\{1\}\inst_ram_wrap.inst_ram
Un-mapped elements :
Overlapping SDs :
Corruption   : Disabled
Power supplies : primary-{ PDI.primary (power,PDI.primary.power) }

Retention policies : 1
-----
1)Name      : ret_PDI
Mapped    : TRUE
Type      : Domain
Save signal : {save_low}
Restore signal : {save_high}
Mapped elements : inst\{1\}\inst_ram_wrap.inst_ram.dout[15:0] inst\{1\}\inst_ram_wrap.inst_ram.din[15:0]
Un-mapped elements :

Isolation policies : 6
-----
1)Name      : iso_PDI_both
Mapped    : TRUE
Type      : Domain
Control signal : {iso_high}
Clamp value  : latch
Mapped elements : inst\{1\}\inst_ram_wrap.inst_ram.dout[0] inst\{1\}\inst_ram_wrap.inst_ram.clk inst\{1\}\inst_ram_wrap.inst_ram.din[0]
Un-mapped elements :

```

- Check the report file for errors. Fix any reported syntax errors for supported UPF commands. You can downgrade some errors (with a DE prefix) to warnings and continue. See [Manipulating Message Display and Reporting, on page 551](#) for details.
- Check the warnings for unsupported UPF commands and messages related to the power domain strategies used. Especially in cases where multiple strategies are defined, check that the intended strategy is implemented for the power domains, and make sure that strategy overrides on instances are as intended.

You can turn on verbose mode to generate a detailed message for every command that is parsed and instrumented in the UPF file. Use the following Tcl option:

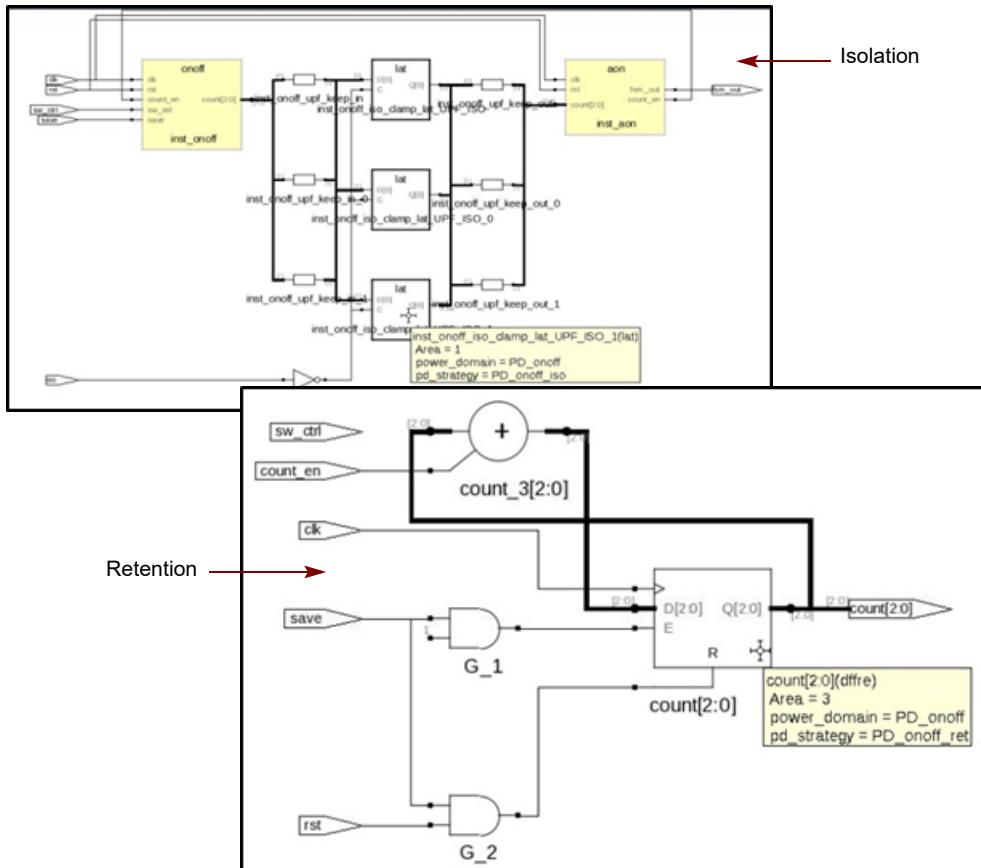
```
option set -upf_verbose 1
```

Downgrade or upgrade applicable error or warning messages as needed to help your work flow.

5. Open the schematic by clicking the icon, and graphically view the isolation and retention elements implemented.

- Open the schematic view with the `view schematic` command, and check the isolation/retention elements.
- Find isolation cells by searching for `*iso_clamp*`, which is part of all inserted isolation cell names.
- Leverage the `power_domain` and `pd_strategy` properties to identify the isolation/retention elements in the schematic. Some examples are shown here:

| Example Tcl Command | Description |
|--|---|
| <code>c_print [find -hier -inst "*" -filter @power_domain==<power_domain name>]</code> | Lists all the cells of the named power domain |
| <code>c_print [find -hier -inst "*" -filter @pd_strategy==<pd_strategy name>]</code> | Lists all the cells of the named pd_strategy |
| <code>c_print [find -hier -inst "*iso_clamp*"] -prop power_domain -prop pd_strategy</code> | Lists all isolation cells, their associated power domains, and strategies |



Verifying Designs with UPF Information

In addition to checking the UPF implementation ([Checking the UPF Implementation, on page 857](#)) you generally verify the power domain sequences in your design by simulating the RTL along with a testbench and the low power intent described in the UPF file. However, if you have implemented your design on an FPGA, do not use this method, but use the following procedure instead.

- Specify the UPF file with the appropriate design phase run command.

You can simulate your design after the pre-partition, pre-map, or map stages. For details, see [Specifying UPF Power Domains, on page 830](#).

2. Use the simulation file generated by the run command to run simulation.

Do not use the UPF file as input to the simulator, because only the logical correctness of the retention and isolation logic can be verified with this file. Use the generated Verilog simulation netlist files instead.

- Locate the appropriate netlist file:

ProtoCompiler Single FPGA

After pre-map *resultBase_premap.vm*

After map *resultBase.vm*

ProtoCompiler Multi FPGA

After pre-partition *resultBase_premap.vm*

After map Top level wrapper: */resultBase.vp*
 Individual partitioned FPGAs: *fpga.vm*

- After mapping, export the file using an export command. For example:

```
export netlist -path path_to_netlist -format verilog
```

3. Run low power tests to verify the UPF implementation.

Converting Gated Clocks

Even though prototyping is done before clock tree synthesis, the clocking scheme in the RTL might still be too sophisticated for an FPGA and must be simplified before the design can be prototyped as an FPGA. Simplify gated clock and generated clock structures by converting them to FPGA-friendly clock schemes.

Gated clocks are clocks where signal propagation is controlled by gates. They are widely used in ASIC design to reduce power when domains are not used. FPGAs have dedicated low-skew clock distribution nets and un-gated clocks, but SoC designs use gated clocks instead of these resources. Generated clocks are clocks created by multiplying, dividing, or time-shifting an existing clock, or by switching inputs or another alignment mechanism.

The functionality to convert gated and generated clocks from the original ASIC designs is called gated clock conversion (GCC). The following sections describe how to use GCC to automatically convert ASIC clocks:

- [Overview of Clock Conversion](#), on page 862
- [Working with Gated and Generated Clocks](#), on page 870
- [Defining Clocks for Gated Clock Conversion](#), on page 872
- [Interpreting Gated Clock Error Messages](#), on page 895

For a discussion about other, general clock issues when moving from ASIC to FPGA, refer to [Converting ASIC Clocks](#), on page 42.

Overview of Clock Conversion

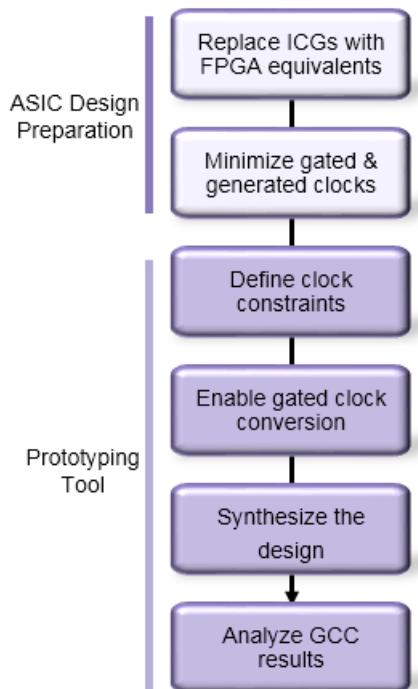
Gated clock conversion or GCC comprises the functionality used to simplify gated clock and generated clock structures and convert them to FPGA-friendly clock schemes. The following topics provide an overview of clock conversion.

- [Gated Clock Conversion Flow](#), on page 863
- [Guidelines for Generated Clocks](#), on page 864
- [Guidelines for Gated Clocks](#), on page 866
- [Guidelines for Integrated Clock Gating Cells \(ICG\)](#), on page 867

Gated Clock Conversion Flow

FPGAs have dedicated low-skew clock distribution nets and un-gated clocks, but SoC designs use gated clocks instead of these resources. The ASIC clock gating logic must be converted before it can use the dedicated clock nets on the FPGA. This process is called gated clock conversion (GCC). GCC creates a logical equivalent without setup and hold violations, by removing the clock gate with the enable signal and routing the enable signal directly to the FPGA logic elements.

The following figure summarizes the steps in the GCC process. For details, see the procedure that follows the figure.



1. Replace integrated clock gating cells (ICGs) with FPGA equivalents.

You can use a netlist editor script to do this. See [Guidelines for Integrated Clock Gating Cells \(ICG\)](#), on page 867 for further information.

2. Follow guidelines for working with generated clocks and gated clocks.

See [Guidelines for Generated Clocks](#), on page 864 and [Guidelines for Gated Clocks](#), on page 866.

3. Enable gated clock conversion and do a preliminary FPGA synthesis run.

Refer to [Working with Gated and Generated Clocks](#), on page 870 for details.
 4. Check the results in the clock conversion report.

The clock conversion report generated after mapping shows the generated and gated clocks that were converted. The tool removes clock gating by moving the gated clock logic from the clock pin of a sequential element to the enable pin.

This example shows a typical clock conversion report:

```

***** START OF CLOCK OPTIMIZATION REPORT *****

Clock optimisation not enabled
2 non-pated/non-generated clock tree(s) driving 3 clock pins(s) of sequential element(s)
3 gated/generated clock tree(s) driving 4 clock pins(s) of sequential element(s)
0 instances converted, 6 sequential instances remain driven by gated/generated clocks

***** Non-Gated/Non-Generated Clocks *****
Clock Tree ID Driving Element Drive Element Type Fanout Sample Instance
Clock$G00004 x_0_gated_clk port 2 00
Clock$G00004 x_0_gated_clk port 1 01 divide_by_x_zin
***** Gated/Generated Clocks *****
Clock Tree ID Driving Element Drive Element Type Fanout Sample Instance
Clock$G00001 xer_gated_zin MPPC 2 00
Clock$G00002 divide_by_2_zin JTAG 2 00
Clock$G00002 and_gated_zin JTAG 2 00

***** END OF CLOCK OPTIMIZATION REPORT *****

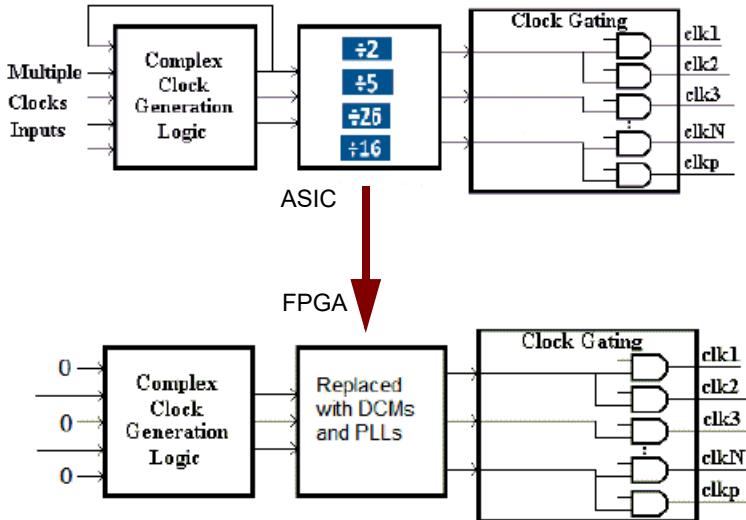

```

For more information about handling GCC errors, see [Interpreting Gated Clock Error Messages](#), on page 895.

Guidelines for Generated Clocks

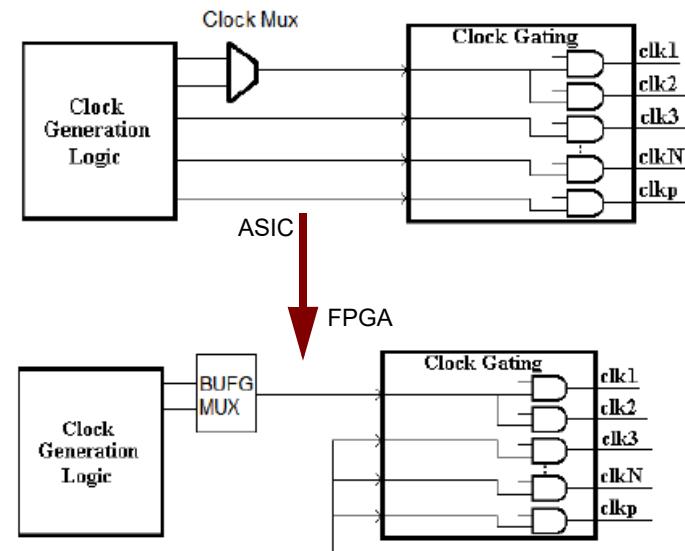
Define clocks that run at sub-multiples of a base clock frequency as generated clocks. Using latch-based clock-gating cells or clock dividers to generate sub-multiple clocks causes those parts of the design to be constrained by the sub-multiple clock frequency instead of the base clock frequency, and this results in faster run times and a smaller area. Designs partitioned across multiple FPGAs additionally benefit from a higher TDM ratio, because the available slack for the signals is larger than when a single clock frequency is used for the whole design.

- Replace generated clocks with equivalent FPGA structures in the target technology, such as clock managers and programmable clock generators like phase-locked loops (PLLs).



- Tie clock inputs that are not going to be used for prototyping to constants. The FPGA prototyping software propagates the constants and optimizes the clock-generating logic, minimizing the clock network. This reduces the number of clocks to be routed inside the FPGAs, and also reduces synthesis and place-and-route runtime.
- When possible, avoid clock muxes. If they cannot be removed or reworked, modify the constraints and declare the clock at the mux output.

Another technique is to replace clock multiplexing logic with FPGA-specific primitives. The following figure shows the clock mux replaced by a Xilinx BUFGMUX, which, along with some additional glitch removal logic, is logically equivalent. The advantage to this approach is that the prototyping software understands these primitives, so timing analysis can be more accurate. Further, it eliminates the need for additional skew logic, because this primitive directly drives global clock resources.

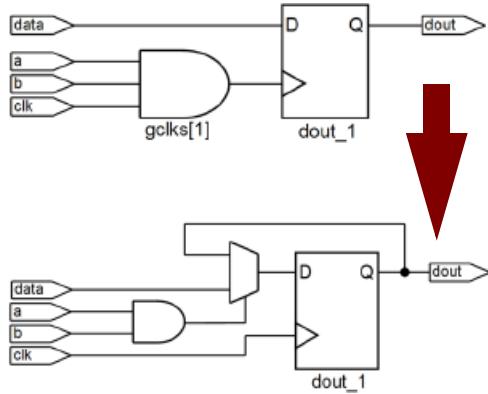


- If you have multiple internally generated clocks for functionality that is not going to be prototyped, tie these clocks to a single external clock source.

Guidelines for Gated Clocks

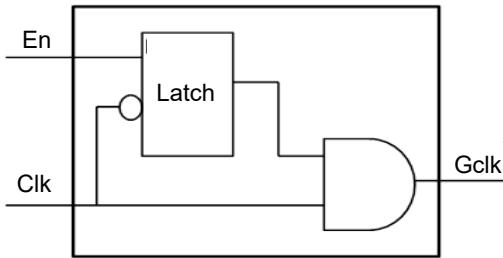
Gated clocks are clocks where signal propagation is controlled by gates. FPGAs have dedicated low-skew clock distribution nets and un-gated clocks, but SoC designs use gated clocks.

To make it possible to use the same RTL for both SoC and FPGA designs, the software provides the functionality to automate the translation. The functionality moves the SoC clock gating from the clock pins of sequential elements to the enable pins. This provides a logically-equivalent FPGA version, without altering the original RTL. For more information about converting gated clocks, see [Converting Gated Clocks, on page 862](#).



Guidelines for Integrated Clock Gating Cells (ICG)

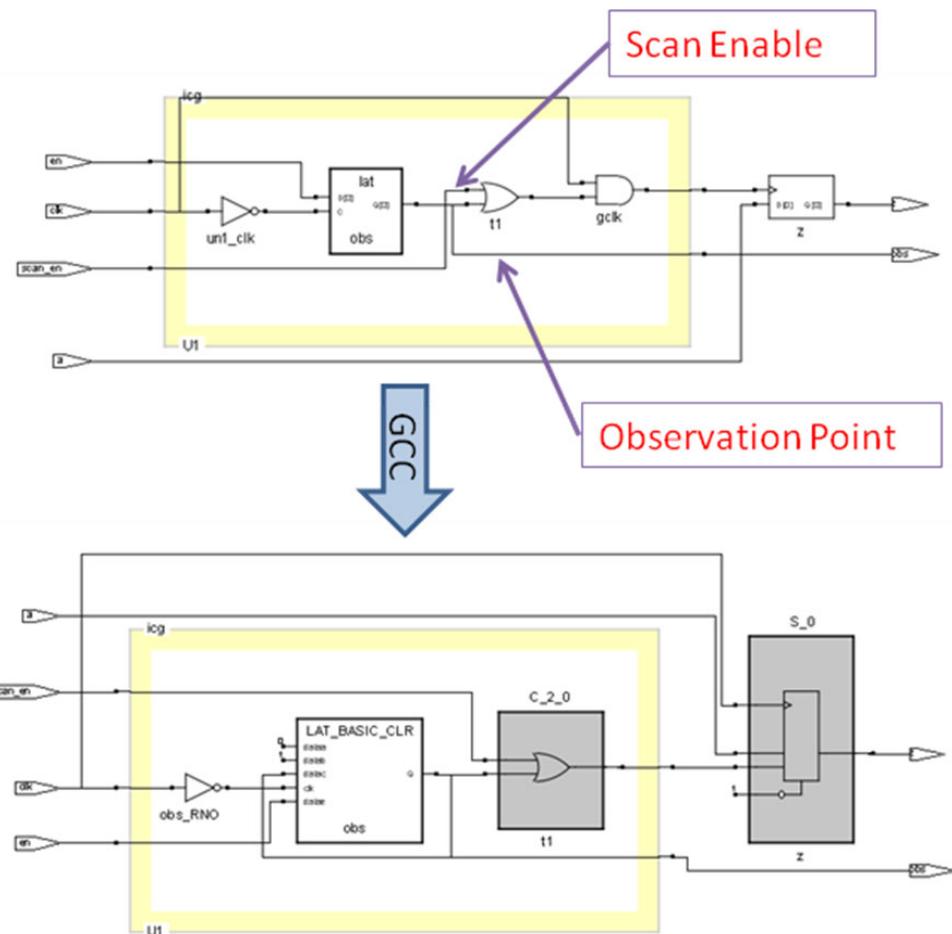
These ASIC cells combine a latch and a clock gate to cut ASIC clock skew. Designs might contain cascading ICGs. Replace these cells with equivalent FPGA logic.



If the ICG cell has ASIC-only pins, such as a scan-enable input or an observation output, remove these pins. These signals force the tool to preserve intermediate signals in the clock-gating circuit so that they cannot be optimized. The following code shows one such example, where the data register is connected directly to the source clock, but the latch and OR gate for the ICG model are part of the enable logic.

```
module icg (clk, en, scan_en, obs, gclk);  
  input clk, en, scan_en;  
  output obs, gclk;  
  wire t0, t1;  
  assign t0 = (!clk) ? en : t0;  
  assign obs = t0;  
  assign t1 = t0 | scan_en;  
  assign gclk = t1 & clk;  
endmodule
```

The following figure shows how the RTL is simplified to remove the intermediate points that are not used in the FPGA. The `scan_en` input and the `obs` observation point output, are tied to a constant, 0. If no other changes are made to the design, this model still fits, because the ports are unchanged. Also, the constant tied to the output propagates, and simplifies downstream logic.



This is the corresponding RTL after modification:

```

module icg (clk, en, scan_en, obs, gclk);
  input clk, en, scan_en;
  output obs, gclk;
  wire t0, t1;
  assign t0 = (!clk) ? en : t0;
  assign obs = 1'b0 /* t0 */;
  assign t1 = t0 | 1'b0 /* scan_en */;
  assign gclk = t1 & clk;
endmodule

```

Working with Gated and Generated Clocks

The following steps describe in detail the procedure needed to identify the gated and generated clocks in your design and set them up for automatic conversion with GCC.

1. Follow the guidelines for gated and generated clocks.

See [Guidelines for Generated Clocks, on page 864](#) and [Guidelines for Gated Clocks, on page 866](#).

2. Define the clocks by setting `create_clock` and `create_generated_clock` constraints.

See [Defining Clocks for Gated Clock Conversion, on page 872](#) for details.

3. Enable gated clock conversion with this command:

```
option set fix_gated_and_generated_clocks 1
```

This command applies globally to the design.

4. To specifically exclude individual clocked elements on critical paths from the global conversion, add a `syn_keep` directive to the input clock net.

You can apply `syn_keep` in a source file or by setting a constraint:

- `syn_keep` set on `gclk` in source file:

```
module and_gate (clk, en, a, z);
  input clk, en, a;
  output reg z;
  wire gclk /* synthesis syn_keep = 1 */;
  assign gclk = en & clk;

  always @ (posedge gclk) z <= a;
endmodule
```

- `syn_keep` set as a constraint on `gclk` in the FDC file:

```
define_attribute {n:gclk} {syn_keep} {1}
```

5. To convert gated-clock and generated-clock circuits created from instantiated FPGA cells, make sure this default setting is in place:

```
option set optimize_ngc 1
```

6. The tool enables flops that connect directly to the clock. The tool does not convert instantiated cells where there are multiple declared clocks. For example, if a MUX is described using instantiated LUT primitives, the tool does not recognize the structure as a MUX, but sees it as multiple clocks feeding a LUT (for example, a 3-input AND gate). Synthesize the design by running compile and map.

For gated clocks, the software converts qualified flip-flops, counters, latches, synchronous memories, and instantiated technology primitives. The tool logically separates the gating from the clock and routes the gating to the clock enables on sequential devices, using the programmable routing resources of the FPGA. The ungated base clock is routed to the clock inputs of the sequential devices using the global clock resources. Because many gated clocks are normally derived from the same base clock, separating the gating from the clock allows a single global clock tree to be used for all gated clocks that reference the same base clock.

7. Analyze results.

- Open the log file and check the Performance Summary section. Check the declared, generated, and derived clocks for issues like incorrect constraints or unsupported structures in clock logic, because gated clock conversion (GCC) is performed on these clocks. GCC is not performed on inferred clocks or on the system clock. Inferred clocks and the system clock are created when no clocks are defined in the fan-in logic of the clock pin of a sequential cell.

| Performance Summary | | | | | | | |
|--------------------------------|---------------------|---------------------|------------------|------------------|--------|---------------------|---------------------|
| Worst slack in design: 8.479 | | | | | | | |
| Starting Clock | Requested Frequency | Estimated Frequency | Requested Period | Estimated Period | Slack | Clock Type | Clock Group |
| clk1 | 100.0 MHz | NA | 10.000 | NA | NA | declared | default_clkgroup |
| clk2 | 111.1 MHz | 1920.5 MHz | 9.000 | 0.521 | 8.479 | declared | default_clkgroup |
| clk3 | 125.0 MHz | NA | 8.000 | NA | NA | declared | default_clkgroup |
| combi gclk_inferred_clock | 1.0 MHz | NA | 1000.000 | NA | NA | inferred | inferred_clkgroup_1 |
| em3_0 | 66.7 MHz | NA | 15.000 | NA | NA | declared | default_clkgroup_1 |
| ff_clk_1 clk_out_derived_clock | 111.1 MHz | 1920.5 MHz | 9.000 | 0.521 | 17.479 | derived (from clk2) | default_clkgroup |

Estimated period and frequency reported as NA means no slack depends directly on the clock waveform

- Check for post-compilation warnings about inferred clocks or the system clock that might affect gated clock conversion. The following are examples of these warnings. In both cases, the clocks have no

specified timing constraints. This may prevent the conversion of gated or generated clocks and adversely impact design performance.

@W: MT529 : Found inferred clock clkName which controls numClkPins sequential elements including *instanceName*.

@W: MT531: Found signal identified as System clock which controls numClkPins sequential elements including *instanceName*.

- For a tabular list of some messages and explanations, see [Interpreting Gated Clock Error Messages, on page 895](#).
- Check the START OF CLOCK OPTIMIZATION REPORT section of the log file for a report on the conversion process.

| 3 – Line Summary | Non-Gated/Non-Generated Clock Tree Table | Gated/Generated Clock Tree Table | | |
|---|--|----------------------------------|-------|----------------------|
| #### START OF PREMAP CLOCK OPTIMIZATION REPORT #####[| | | | |
| 3 non-gated/non-generated clock tree(s) driving 126030 clock pin(s) of sequential element(s) 4 gated/generated clock tree(s) driving 248996 clock pin(s) of sequential element(s) 123802 instances converted, 248996 sequential instances remain driven by gated/generated clocks | | | | |
| ===== Non-Gated/Non-Generated Clock Tree ID Driving Element Drive Element Type Fanout Sample Instance ===== | | | | |
| ClockId 0_0 | il.inst.clkout1_buf.0 | BUFG | 62379 | genblk1\[4\]\.cl.ico |
| ClockId 0_1 | xjtag_tck | port | 1272 | genblk1\[4\]\.cl.ico |
| ClockId 0_2 | il.inst.clkout2_buf.0 | BUFG | 62379 | genblk1\[5\]\.cl.ico |
| ===== Gated/Generated Clock Tree ID Driving Element Drive Element Type Unconverted Fanout Sample Instance ===== | | | | |
| ClockId 0_4 | xck_gated[3].OUT | and | 62249 | genblk1\[3\]\.cl.ico |
| ClockId 0_6 | xck_gated[2].OUT | and | 62249 | genblk1\[2\]\.cl.ico |
| ClockId 0_8 | xck_gated[1].OUT | and | 62249 | genblk1\[1\]\.cl.ico |
| ClockId 0_10 | xck_gated[0].OUT | and | 62249 | genblk1\[0\]\.cl.ico |
| ##### END OF CLOCK OPTIMIZATION REPORT ##### | | | | |

See [Interpreting Gated Clock Error Messages, on page 895](#) for a description of conversion-specific error messages you might see.

Defining Clocks for Gated Clock Conversion

Follow these guidelines to define gated and generated clocks in the design before you synthesize it.

1. Make sure that the gated clocks you define satisfy these requirements:

- Gated clock logic must consist only of combinational logic. Derived clocks that are the output of registers are not converted.
- Additionally, the combinational logic must satisfy the conditions described below. See *Combinational Logic and GCC Examples*, on page 879.

Condition 1 For at least one set of gating input values, the value output for the gated clock must be constant and not change as the base clock changes.

Condition 2 For at least one value of the base clock, changes in the gating input must not change the value output for the gated clock.

- The sequential primitive clocked by the gated clock must be a supported object. The tools support gated-clock conversion for most sequential primitives.
- If you have black boxes driven by gated clocks, use the `syn_force_seq_prim`, `syn_isclock`, and `syn_gatedclk_en` directives to define the clock and clock enable inputs to the black box. The tool does not check the functionality of the contents of the black box. See *Gated Clock Definitions for Black Boxes*, on page 881 for examples.

2. To identify a net as a clock, specify a period or frequency constraint for either the gate or the clock using the `create_clock` constraint.
 - Refer to the general guidelines in *Converting ASIC Clocks*, on page 42.
 - Set the constraint in the FDC constraint file, using syntax like this:

```
create_clock -name {clk} -freq 10.000 -clockgroup default_clkgroup
```

3. Define the relationship between generated clocks and their sources by specifying `create_generated_clock` constraints.

Do the following to specify the constraint:

- Specify the master clock source (a clock source pin in the design). Use the schematic viewer to identify the names, and copy and paste them into the constraint.
- Specify the frequency for the generated clocks as a multiplication or division factor (`-multiply_by` | `-divide_by`). If you use `-multiply_by`, use `-duty_cycle` to specify the percentage of the high pulse width to use for the generated clock.

- Specify whether the generated clock signal should be inverted (-invert).
- Specify the clock edges (-edges).

The example below shows some generated clock constraints. See [GCC Examples for Generated Clocks, on page 885](#) for additional examples, and [create_generated_clock, on page 292](#) in the *Command Reference* for details about the command syntax.

```
set CLK 48
create_clock -name CLK -period $CLK [get_ports CLK]
create_generated_clock -name CLK1 -add [get_nets CLKOUT1]
    -source [get_ports CLK] -master_clock [get_clocks CLK] -edges {1 2 5}
create_generated_clock -name CLKPROCBUS -add [get_nets
    u_Proc.CLKOUT] -source [get_ports CLK] -master_clock [get_clocks
    CLK]-edges {1 2 5}
create_generated_clock -name CLK2 -add [get_nets CLKOUT2] -source
    [get_ports CLK] -master_clock [get_clocks CLK] -edges {1 2 9}
set_clock_groups -asynchronous {CLK CLK1 CLKPROCBUS CLK2}
```

Follow these guidelines when setting generated clock constraints:

- Define gated clocks at the nodes to be connected to the clock pins of the sequential cells.
- Identify only one input to the combinational logic for the gated clock as a base clock. If you do not specify an explicit constraint on the clock net or if you set a global frequency constraint, the clocks will not be converted during synthesis.
- Define one clock for every fan-in cone clock pin on sequential devices. If you do not do this, the tool may not be able to trace the clock source. Typically, the tool traces the clock pin back to the first multiple-input gate. If it is faced with a choice of inputs at this point, the tool infers a clock at the output of that cell. The inferred clocks are not user-defined, so the tool does not convert them.

If you define more than one clock per clock pin, the tool will not perform gated clock conversion.

- Set this constraint if a clock circuit contains both a clock generator and a gating element, to define the relationship.
4. To provide more meaningful names for automatically-generated clocks, use `create_generated_clock`.

If you have defined the associated master clock, the tool automatically generates clocks at the output of clock-modifying blocks such as MMCMs and PLLs and names them. To rename them, use `create_generated_clock` as shown in example below. Renamed clocks are forward-annotated in the xdc file, and Vivado recognizes the renamed clocks.

```
create_generated_clock -name {mem_clk} [get_pins  
{mmcm0.CLKOUT0}]
```

5. To convert datapath latches that are driven by generated clocks and are set or reset by asynchronous signals, follow these steps:
 - Ensure that the combinational logic is driven by flip-flops.
 - Make sure that the input flip-flops do not have an active set or reset (an active-low reset must be tied high, or an active-high set must be tied low). The rules apply to all the input flip-flops in the cone. In addition, all input flip-flops must be driven by the same edge of the same clock.
 - Set the option to force asynchronous flip-flops to be converted:

```
option set force_async_genclk_conv 1
```

The tool does not convert gated or generated clocks if the driving flip-flop has asynchronous sets or resets, unless the flip-flop or datapath latch being converted also has asynchronous controls tied to the same net as the driving flip-flop. With the option set as shown above, the tool ignores asynchronous sets or resets on clocks generated by flip-flops and converts its loads. See [Example 4: Conversion with Asynchronous Set or Reset Signals, on page 887](#).

Note that forcing gated clock conversion with this option can miss edges introduced by the asynchronous controls that the original circuit detected. It might also trigger on invalid edges that were masked by the asynchronous controls in the original circuit.

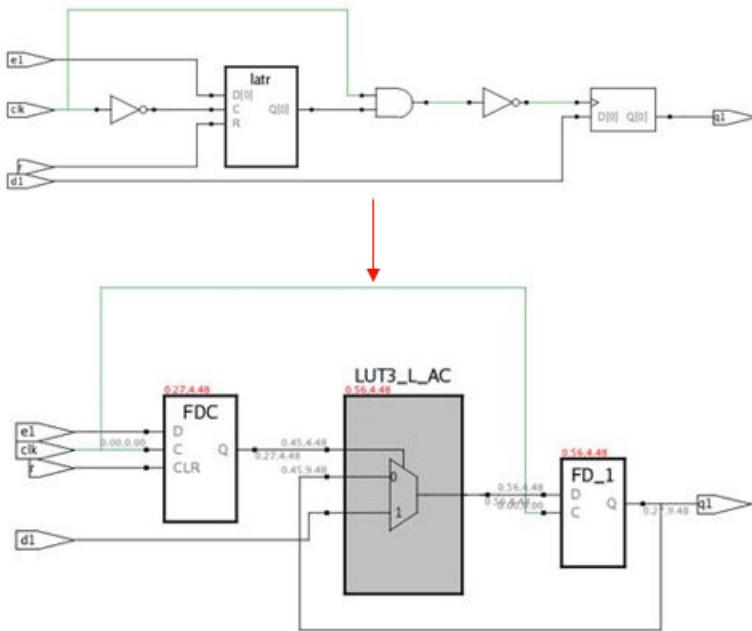
6. Stability latches in the Integrated Clock Gating (ICG) circuit that are moved to the enable path of sequential loads after gated and generated clock conversion (GCC) can cause timing violations for Vivado place and route. Therefore, in the pre-map stage the synthesis tool removes the stability latches in the ICG cell based on the following:
 - A latch on the clock path is converted to a mux and flip-flop.

- The mux select pin is connected to 1 or 0, depending on the sensitive clock edges of the sequential loads. For a rising clock edge, the mux selects 0. For a falling clock edge, the mux selects 1. Then constant propagation is implemented through the mux.

To convert ICG latches that are driven by gated and generated clocks and are set or reset by asynchronous signals, follow these steps:

- Ensure that the ICG latch signals that drive the gated and generated clocks are either asynchronous set or reset. If both asynchronous set and reset signals for the ICG latch are used, then the latch is seen as an unsupported load and will not be removed.
- Make sure that the ICG latch drives only a positive (rising) or negative (falling) edge flip-flop. Do not mix rising and falling clock edges to the loads of the flip-flop.
- Set the option to force asynchronous flip-flops to be converted:
`option set -force_async_genclk_conv 1`

The following example shows an ICG latch with asynchronous reset signals driving a negedge load and what occurs after GCC optimizations.



ICG Latch Removal Summary:

Number of ICG latches removed: 1

Number of ICG latches not removed: 0

For details and more examples, see [Integrated Clock Gating \(ICG\) Stability Latch Removal Examples](#), on page 887.

7. To specify clock latency, follow these steps:

- Start with a MMCM output pin that you know will be driving a global net, and declare a `create_generated_clock` constraint for it.
- Set the `haps_global_clock` attribute to define clock latency.

For example:

```
create_generated_clock -name {gen_clk} -source [get_ports {clk}] -divide_by 2
    [get_pins {U_clk_gen.MMCM_i.CLKOUT0}]
define_attribute [get_pins {U_clk_gen.MMCM_i.CLKOUT0}] { haps_global_clock } {1}
```

Defining Generated Clocks with `create_generated_clock`

Define the relationship between generated clocks and their sources with `create_generated_clock` constraints. The `create_generated_clock` constraint provides a way to specify timing for gated and generated clocks. Use it with the `create_clock` constraint to modify timing.

1. Specify the `create_generated_clock` constraint in the FDC file or through the Clocks tab in the SCOPE GUI.

Set the `create_generated_clock` constraint if a clock circuit contains both a clock generator and a gating element, to define the relationship. See [`create_generated_clock`, on page 292](#) in the *Command Reference* for details about the command syntax referred to in this procedure.

2. In the constraint, define a clock source pin as the master clock source (`-source`).

Use the schematic viewer to identify the source pin names, and copy and paste them into the constraint.

3. Specify the frequency for the generated clocks with `-multiply_by` | `-divide_by`.

The frequency is set as a multiplication or division factor of the source clock, using the appropriate argument. If you use `-multiply_by`, also use `-duty_cycle` to specify the percentage of the high pulse width to use for the generated clock.

4. Specify whether the generated clock signal should be inverted (`-invert`).
5. Specify the clock edges (`-edges`).

The example below shows some generated clock constraints. See [*GCC Examples for Generated Clocks*, on page 885](#) for additional examples.

```
set CLK 48
create_clock -name CLK -period $CLK [get_ports CLK]
create_generated_clock -name CLK1 -add [get_nets CLKOUT1]
    -source [get_ports CLK] -master_clock [get_clocks CLK] -edges {1 2 5}
create_generated_clock -name CLKPROCBUS -add [get_nets
    u_Proc.CLKOUT] -source [get_ports CLK] -master_clock [get_clocks
    CLK]-edges {1 2 5}
create_generated_clock -name CLK2 -add [get_nets CLKOUT2] -source
    [get_ports CLK] -master_clock [get_clocks CLK] -edges {1 2 9}
    set_clock_groups -asynchronous {CLK CLK1 CLKPROCBUS CLK2}
```

6. To provide more meaningful names for automatically-generated clocks, use `create_generated_clock`.

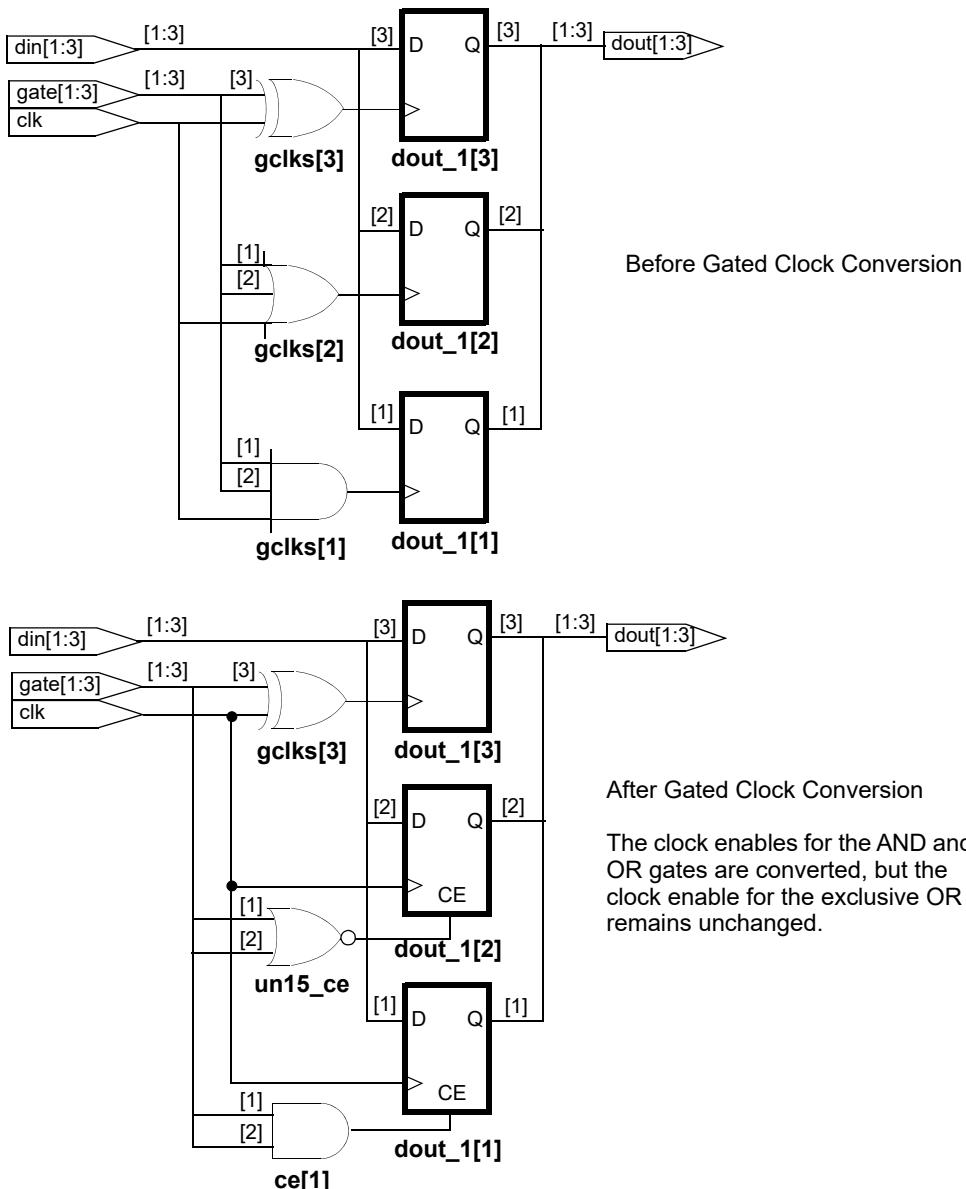
If you have defined the associated master clock, the tool automatically generates clocks at the output of clock-modifying blocks such as MMCMs and PLLs and names them. To rename them, use `create_generated_clock` as shown in example below. Renamed clocks are forward-annotated in the xdc file, and Vivado recognizes the renamed clocks.

```
create_generated_clock -name {mem_clk} [get_pins
{mmcm0.CLKOUT0}]
```

Combinational Logic and GCC Examples

The correct logic format requirements are illustrated with the simple gates shown in the following figures. When the software synthesizes the gated clocks defined, it converts clock enables for the AND and OR gates, but does not convert the exclusive-OR gate shown in the second figure. The following table explains the details.

| Gate | Condition 1 | Condition 2 | Result |
|-----------------|--|--|---|
| AND gclks[1] | Met If either gate[1] or gate[2] is 0, then gclks[1] is 0, independent of the value of clk. | Met If clk is 0, then gclks[1] is 0, independent of the values of gate[1] and gate[2]. | gclks[1] is successfully converted to clock-enable format. |
| OR gclks[2] | Met If either gate[1] or gate[2] is 1, then gclks[2] is 1 independent of the value of clk. | Met If clk is 1, then gclks[2] is 1 independent of the value of gate[1] or gate[2] which satisfies the second condition | gclks[2] is successfully converted to the clock-enable format |
| XOR gclks[3] | Not met gclks[3] continues to toggle, regardless of the value of gate[3] | | Exclusive OR, gclks[3], is not converted. |



Gated Clock Definitions for Black Boxes

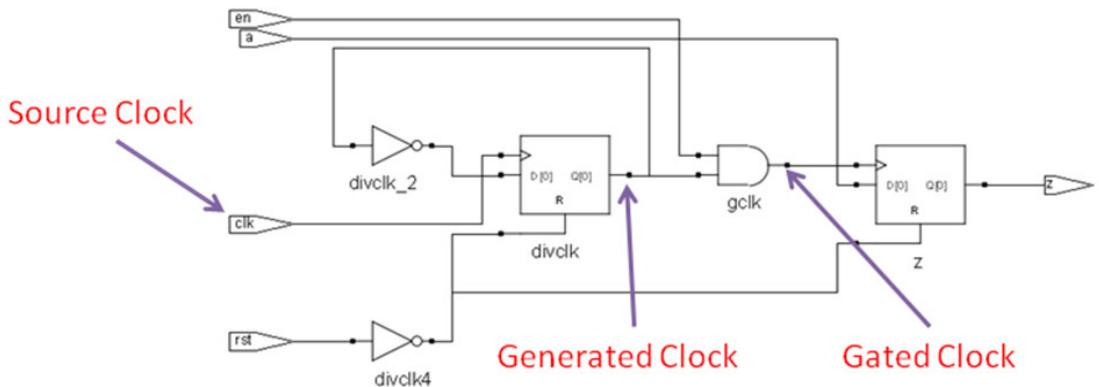
The following Verilog example specifies the `syn_force_seq_prim`, `syn_isclock`, and `syn_gatedclk_clock_en` directives, ensuring that the gated clocks that drive the black box get converted during FPGA synthesis.

Verilog

```
module bbe (ena, clk, data_in, data_out)
/* synthesis syn_black_box */
/* synthesis syn_force_seq_prim="clk" */
;
input clk
/* synthesis syn_isclock = 1 */
/* synthesis syn_gatedclk_clock_en="ena" */;
input data_in,ena;
output data_out;
endmodule
```

GCC Clock Constraint Examples

This is the sample design:

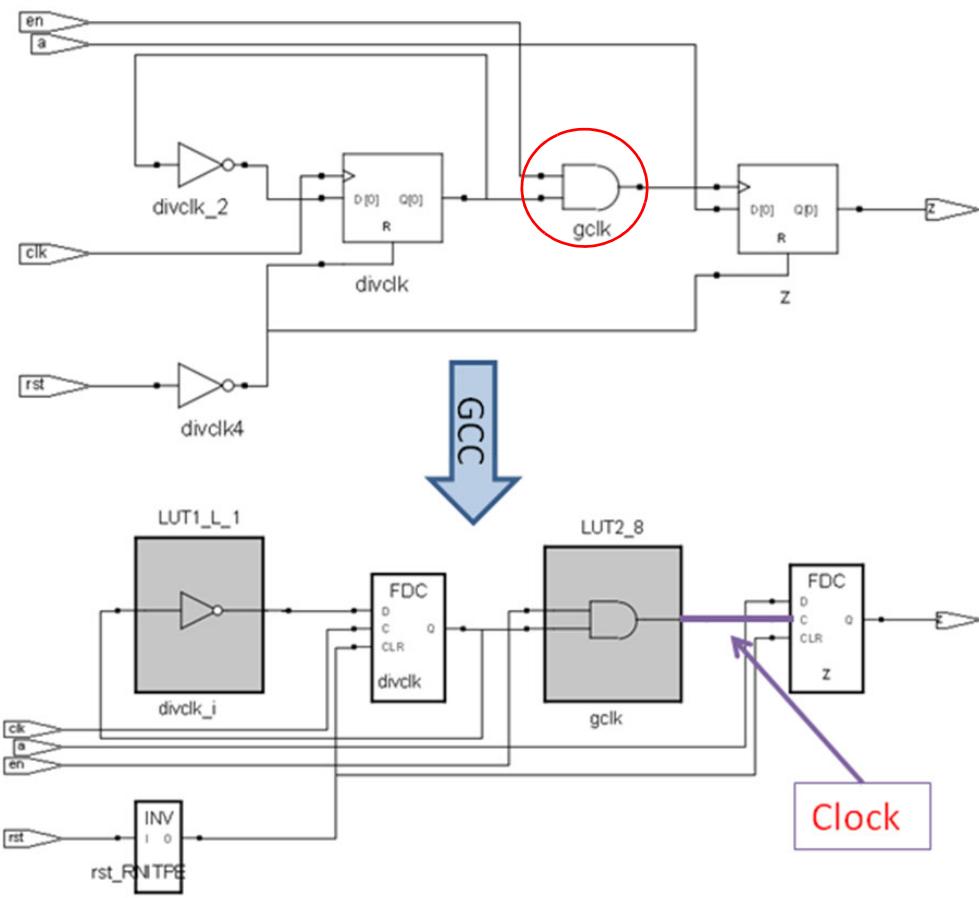


GCC with `create_clock` Definition for Source Clock Only

This constraint defines the source clock:

```
create_clock -name clk [get_ports clk] -period 10
```

`get_ports` identifies `clk` as a port. If only the source clock is defined, the tool cannot determine which input of the AND gate is the clock and which input is the enable, and the tool does not perform GCC:

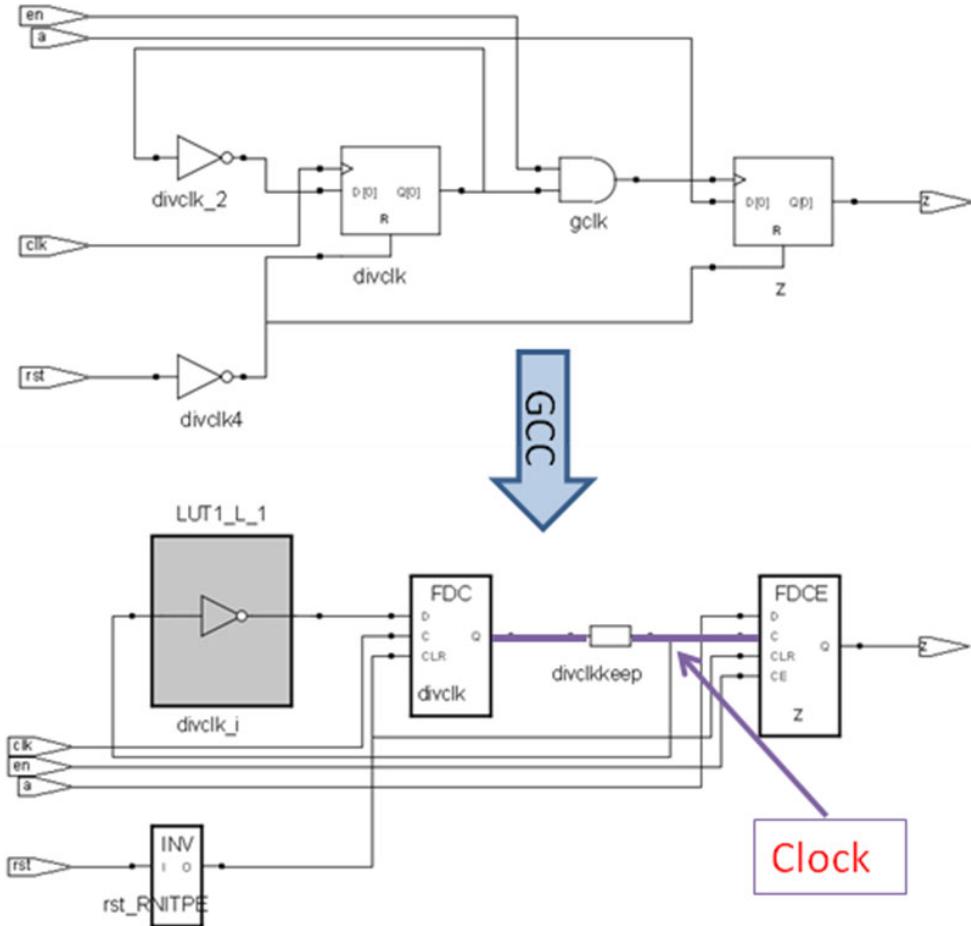


GCC with create_clock Definitions for Source and Generated Clocks

This example adds a second constraint, and define a source clock and a generated clock:

```
create_clock -name clk [get_ports clk] -period 10  
create_clock -name divclk [get_nets {divclk}] -period 20
```

In the second constraint, `get_nets` identifies `divclk` as a net. Independent clock constraints now define the source and generated clocks, so the tool can extract which input of the AND gate is the clock. However the tool still does not have information about the relationship between the source and generated clocks, so it converts clock gate and data register ‘Z’ to an enable flip-flop (FDCE), and connects that to the generated clock:

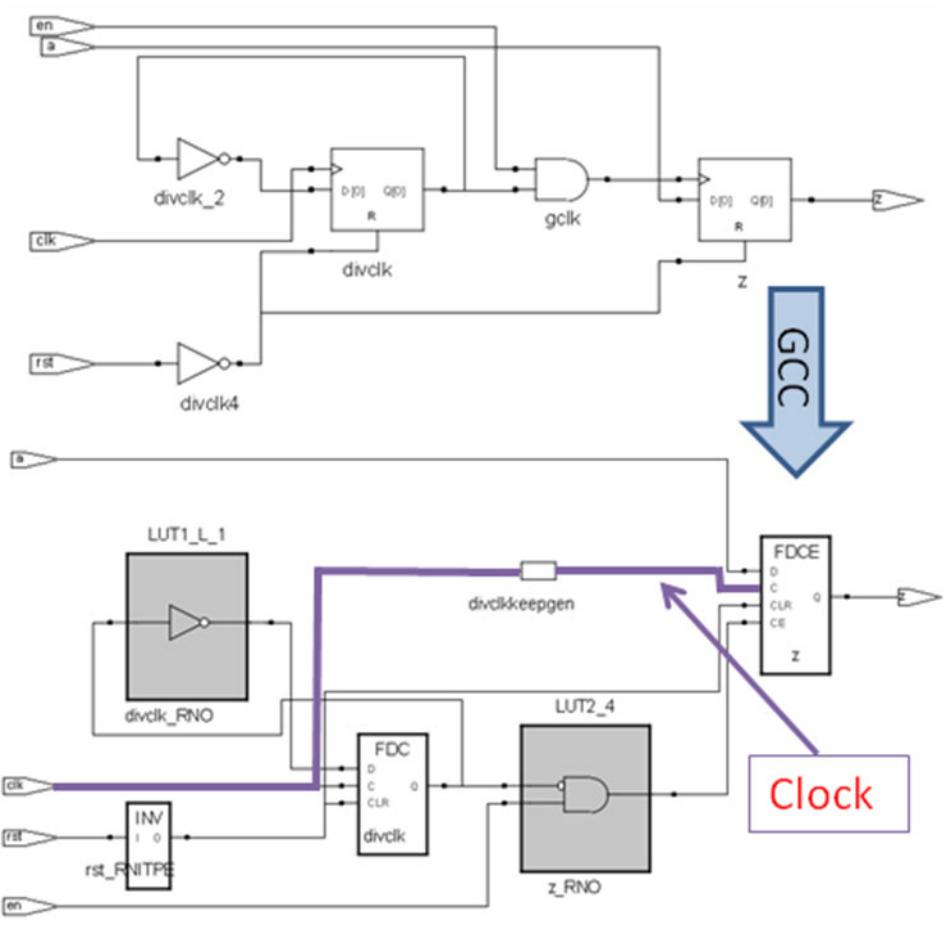


GCC with Complete Definitions for Source and Generated Clocks

This example uses `create_clock` for the source clock, as in the previous examples, and `create_generated_clock` for the generated clock:

```
create_clock -name clk [get_ports clk] -period 10
create_generated_clock -name divclk [get_nets {divclk}]
-source [get_ports clk] -divide_by 2
```

This combination of constraints defines the clock input to the AND gate for the tool, as well as the relationship between the source and generated clocks. The GCC process converts the entire clock circuit to an enable on the Z register, and connects the Z register clock pin directly to the source clock:

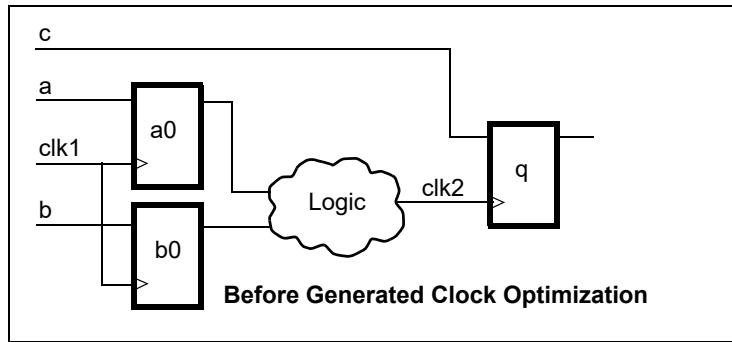


GCC Examples for Generated Clocks

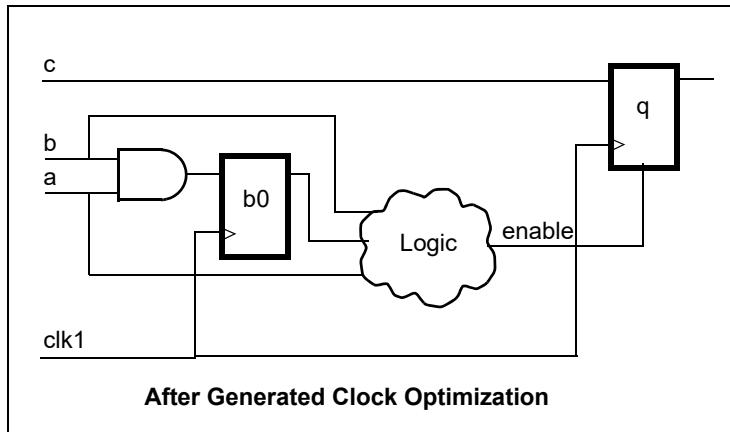
These examples show how gated clock conversion works with generated clocks.

Example 1: Generated Clock from Combinational Logic

The q flip-flop is driven by a generated clock that originates from combinational logic that is driven by the ao and bo flip-flops. These flip-flops are driven by the initial clock (clk1).

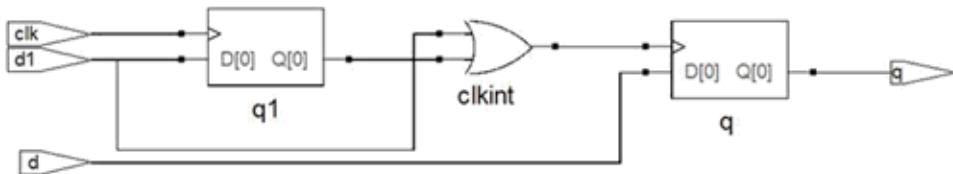


The GCC process replaces q with an enable flip-flop that is clocked by the initial clock and enabled by combinational logic based on the a and b inputs:



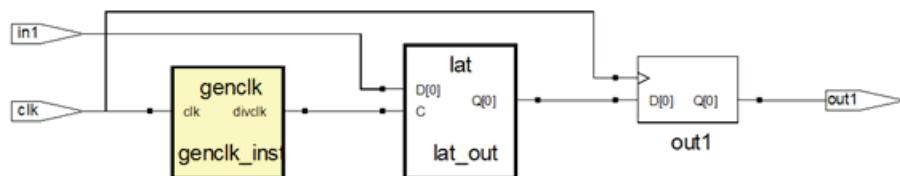
Example 2: Generated Clock Followed by a Gated Clock

This example has a register that is clocked by a generated clock, then followed by a gated clock. If the generated clock constraint also includes the enable, the register might be optimized away during synthesis and replaced with a constant value. The GCC process converts the generated and gated clock logic into a clock and an enable; the enable may never go active.

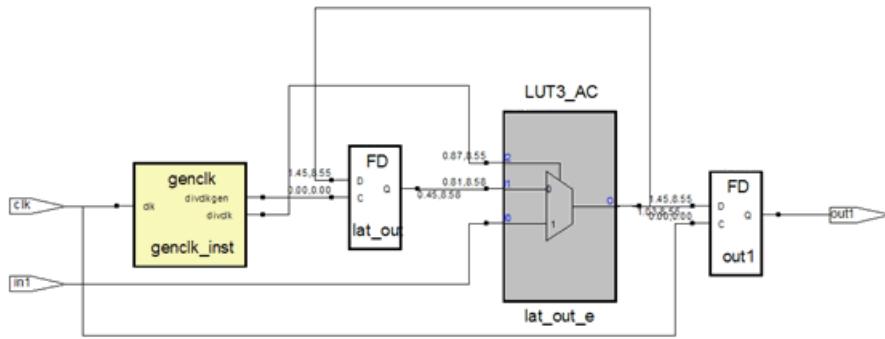


Example 3: Datapath Latch Conversion

This example shows a latch on the datapath that is clocked by a generated clock.



GCC converts the circuit to a flip-flop and multiplexer:

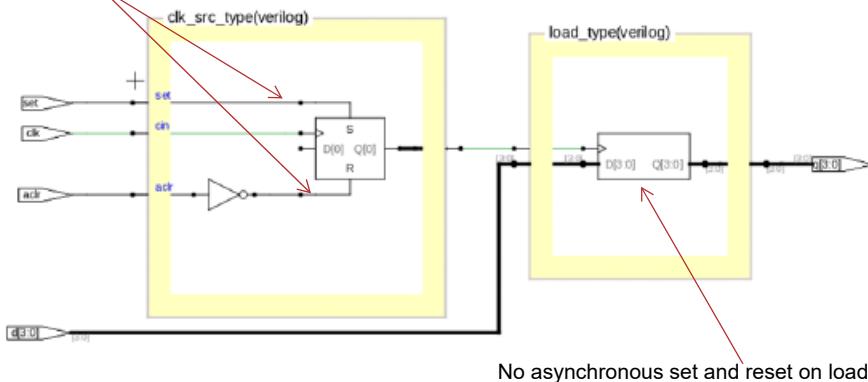


Example 4: Conversion with Asynchronous Set or Reset Signals

In this example, there are asynchronous set and reset signals on the clock generated from the flip-flop, but the load does not have asynchronous set and reset signals. By default, the tool does not run GCC in this scenario.

However, when `force_async_genclk_conv` is set to 1, GCC handles this generated clock scenario and converts the circuit to a flip-flop and multiplexer.

Asynchronous set and reset signals on generated clock

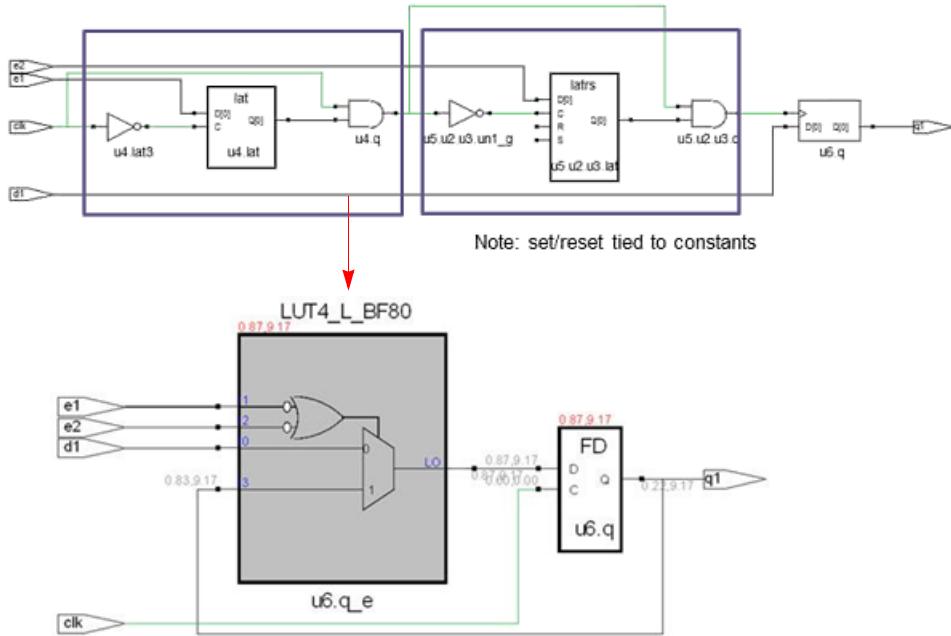


Integrated Clock Gating (ICG) Stability Latch Removal Examples

After GCC, the mux is removed and the flip-flop is retained or removed. The following examples show how the ICG stability latches are implemented for various ICG conditions.

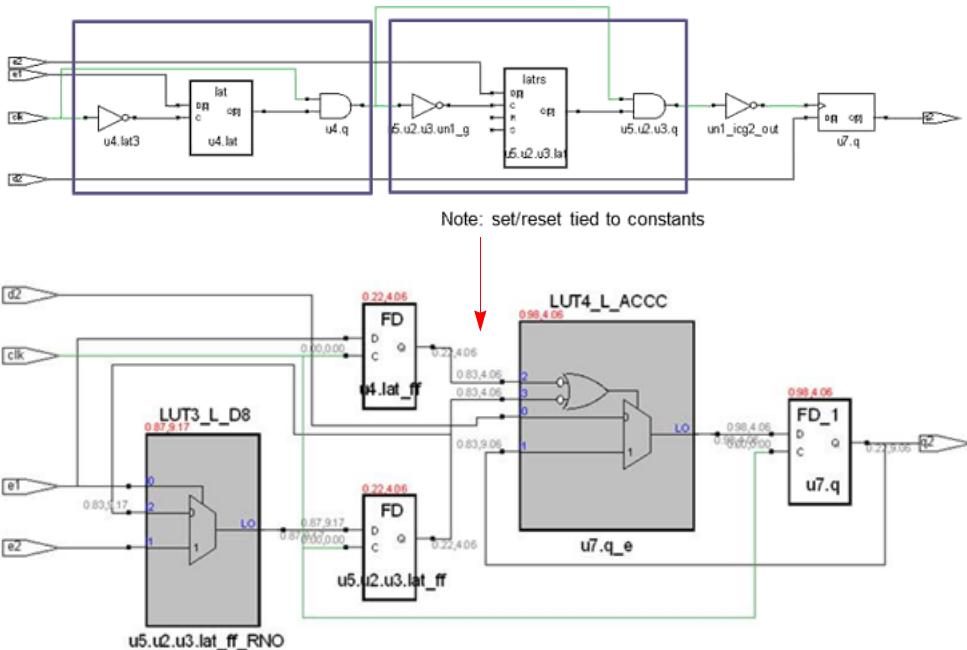
Example 1

This example shows a cascading ICG that drives a positive edge flip-flop and what occurs after GCC optimizations.



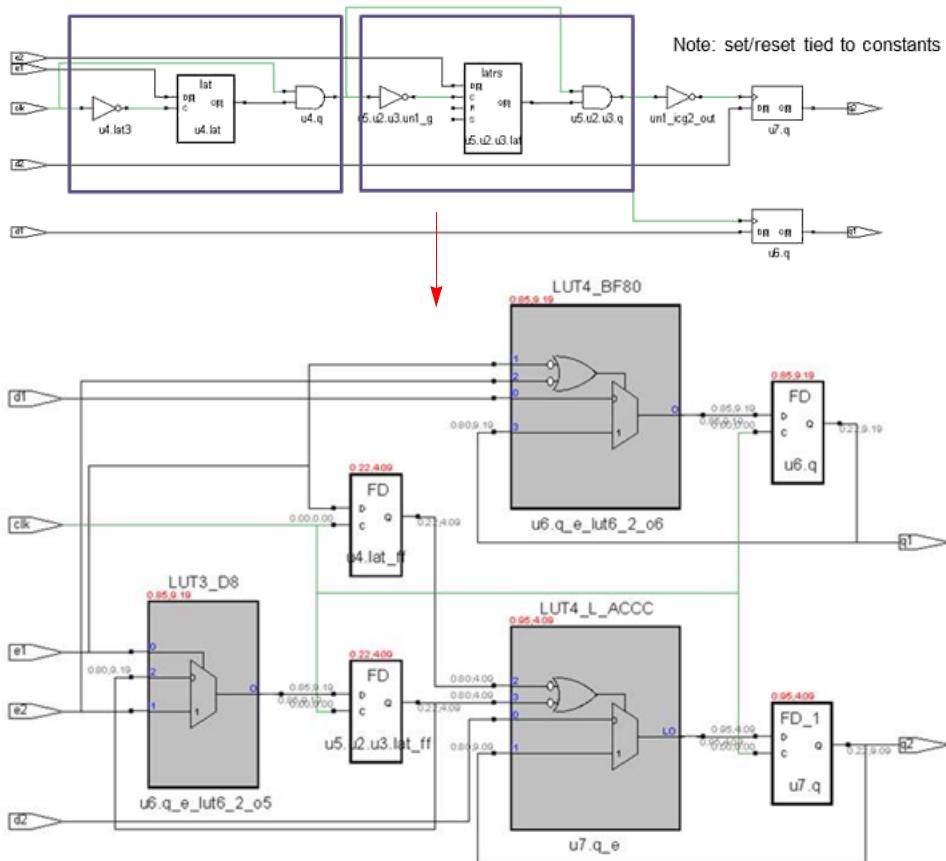
Example 2

This example shows a cascading ICG that drives a negative edge flip-flop and what occurs after GCC optimizations.



Example 3

This example shows a cascading ICG that drives a positive and negative edge flip-flop and what occurs after GCC optimizations.



You can check the ICG Latch Removal Summary in the pre-map section of the log file for the:

- Number of ICG latches removed
- Number of ICG latches not removed
- Following message that is generated:

@N: ICG latches not removed are assigned the property `icg_retain` set to 1. Use Tcl find to locate the latches with this property.

In the ICG Latch Removal Summary, note that:

- ICGs that drive black boxes are not counted in the Number of ICG latches not removed.
- Latches that are not removed can be optimized away later, after GCC. These latches removed by GCC are not counted in the ICG Latch Removal Summary.

Unsupported Loads for ICG Latches

The following examples provide conditions why ICG stability latches are not removed.

Example 1: Unsupported latches not considered to be ICG latches

Latches *not* considered to be ICG latches, do not get reported in the ICG Latch Removal Summary.

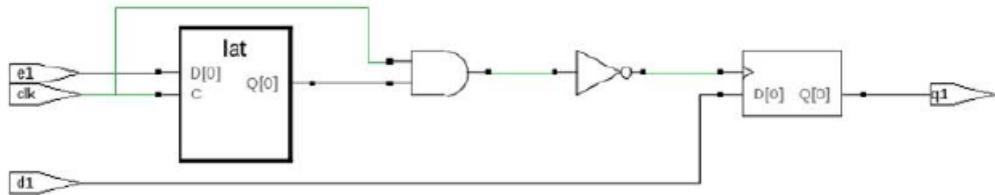
```

Report: top (icg_like_type1_fd_1)
@N:MF248 : | Running in 64-bit mode.
@N:MF666 : | Clock conversion enabled
Design Input Complete (Real Time elapsed 0h:00m:00s; CPU Time
@N:FX1020 : | Library default initial value being ignored on
Mapper Initialization Complete (Real Time elapsed 0h:00m:00s;
Start loading timing files (Real Time elapsed 0h:00m:00s; CPU
Finished loading timing files (Real Time elapsed 0h:00m:00s; C
Reading chip information from file [/remote/sbg_rel/unix/lates
Reading Xilinx I/O pad type table from file [/remote/sbg_rel/u
Reading Xilinx Rocket I/O parameter type table from file [/rem

ICG Latch Removal Summary:
Number of ICG latches removed: 0
Number of ICG latches not removed: 0
ICG Latch Removal takes time: 0.000020 [s].

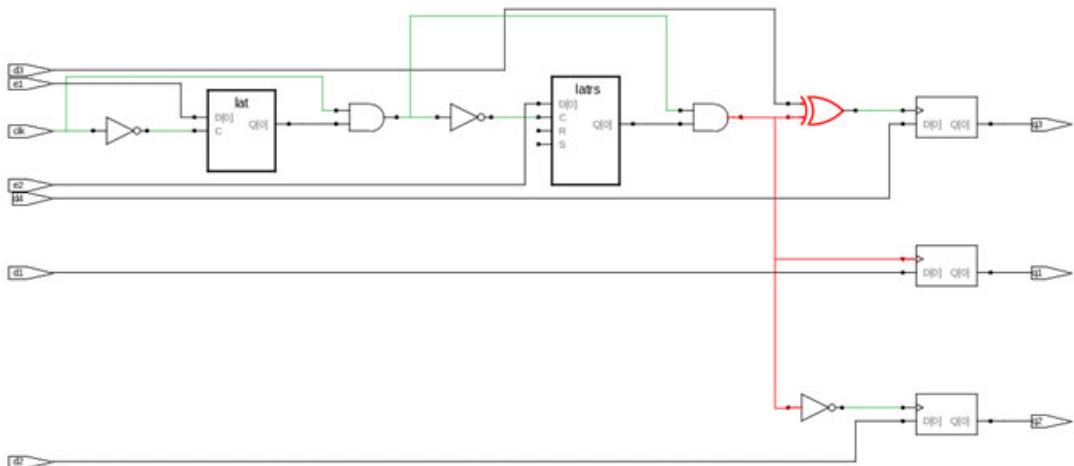
```

The following figure might include an unsupported latch, where an incorrect phase of the clock is connected to the latch, such that it is not removed.



Example 2: ICGs driving unsupported loads

In the following figure, the ICG circuitry drives an unsupported load for an XOR gate. This results in the ICG circuitry not being removed.



The following warning messages are written to the log file:

@W:MF760: | ICG Latch Removal: Unable to remove ICG latch u4.lat because it drives unsupported instance gclk.

@W:MF760: | ICG Latch Removal: Unable to remove ICG latch u5.u2.u3.lat because it drives unsupported instance gclk.

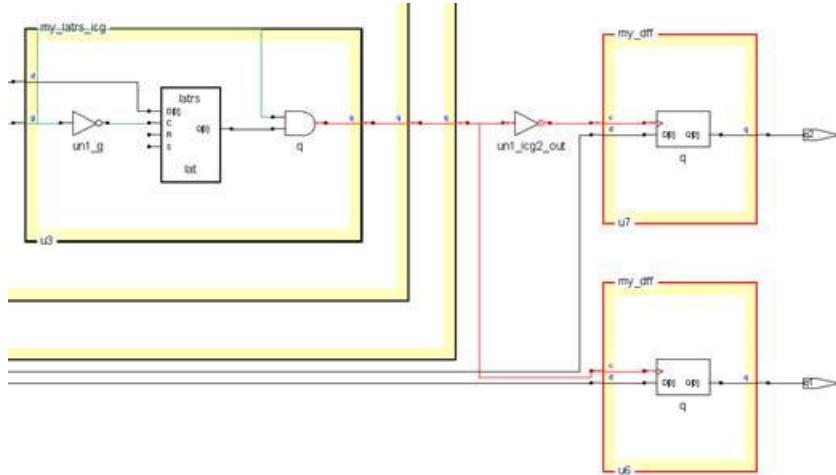
ICG Latch Remove Summary:

Number of ICG latches removed: 0

Number of ICG latches not removed: 2

Example 3: ICGs driving compile points for loads with mixed clock edges

In this scenario, posedge and negedge register loads are inside hard compile points. Since the registers have two different clock edges and are inside compile points, the ICG circuitry is not removed.

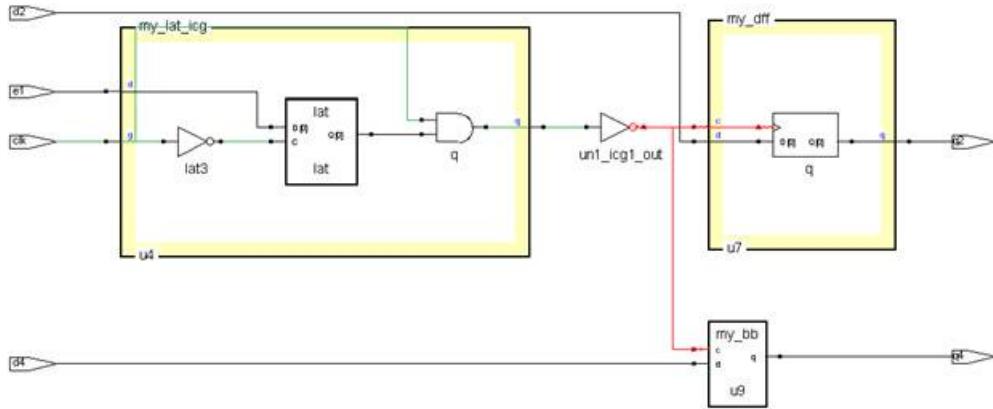


The following warning message is written to the log file:

```
@W:MF758: | ICG Latch Removal: Unable to remove ICG latch u5.u2.u3.lat
because it drives a compile point and rising/falling clock edges
```

Example 4: ICGs driving both black boxes and registers

For this scenario, the ICG circuitry driving the black box gets replicated and is not removed. However, the original ICG circuitry that drives `u1.q` is removed.

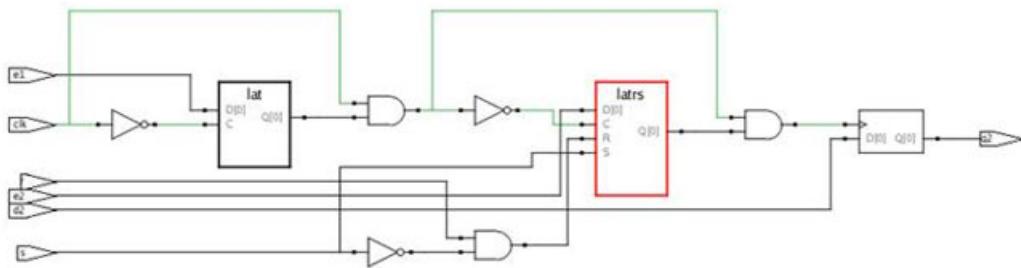


The following warning message is written to the log file:

@N:MF761: | ICG Latch Removal: Blackbox load found on ICG latch u4.lat; latch replicated and not removed for blackbox load.

Example 5: ICG latch with asynchronous set and reset signals

For this example, the ICG latch cannot be removed because the ICG latch has asynchronous set and reset signals.



ICG Latch Removal Summary:

Number of ICG latches removed: 0

Number of ICG latches not removed : 1

The following warning message is written to the log file:

@W:MF759: | ICG Latch Removal: Unable to remove ICG latch u4.lat because it drives non-stability latch u5.lat

Interpreting Gated Clock Error Messages

This procedure describes how to analyze gated clock conversion (GCC) and troubleshoot the design based on messages reported after the run.

1. Check the log file and GCC report for the results of gated clock conversion at different points in the design flow.

You can check the design before GCC optimization for a baseline. After GCC, the log file reports the number of keepbufs and constraints as well as an ICG report. The post-GCC report summary contains details of the GCC implementation.

2. Check the messages reported in the Gated/Generated Clocks section of the clock conversion report.
 - The general procedure is to trace the clock path and check that the clocks are properly constrained and that the clock structure is one that is supported for gated clock conversion.
 - For information how to deal with some of the specific situations reported, see the following table.

The following table describes some gated clock conversion error messages reported in the Gated/Generated Clocks section of the clock conversion report, in alphabetical order.

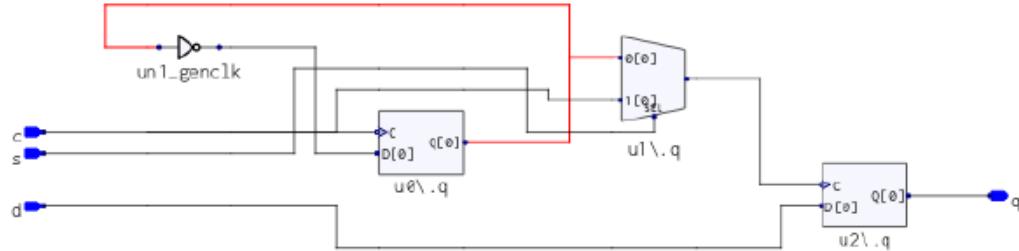
| Error Message | Explanation |
|---|---|
| Asynchronous set/reset mismatch prevents generated clock conversion | Unshared asynchronous signals have been detected between a FF-derived clock circuit and its sequential load. |
| Can't determine input clock driver | Self-explanatory. Specify the driver. |
| Clock conversion disabled | Gated clocks have been detected, but clock conversion is not enabled. |
| Clock from clock mux | The mux is fed by a top-level clock on one input and a generated clock (derived from same top-level clock) on the second input. See Clock from Clock Mux , on page 898. |
| Clock from gated/generated/unconstrained technology primitive | There is a gated or generated clock upstream from the instance. See Clock from Gated/Generated/Unconstrained Technology Primitive , on page 898. |

| Error Message | Explanation |
|--|---|
| Clock from generated clock directive | There is a <code>create_generated_clock</code> constraint on a mux output. See Clock from Generated Clock Directive , on page 899. |
| Clock on mux select line | A clock signal is driving a select pin on a mux. See Clock on Mux Select Line , on page 899. |
| Clock propagation blocked by fixed hierarchy | Clock property has been blocked by a fixed hierarchy which prevents clock conversion from propagating upstream. |
| Clock propagation blocked by hard hierarchy | Clock property has been blocked by a hard hierarchy which prevents clock conversion from propagating upstream. |
| Clock propagation blocked by locked hierarchy | Clock property has been blocked by a locked hierarchy which prevents clock conversion from propagating upstream. |
| Clock propagation blocked by <code>syn_keep</code> | Clock property has been blocked by a <code>syn_keep</code> property which prevents clock conversion from propagating upstream. |
| Clock source is constant | Self-explanatory. |
| Clock source is invalid for GCC | A general introductory note that precedes other messages with specific details. |
| Combinational loop in clock network | Self-explanatory. Fix the combinational loop. |
| Derived clock on input | There is an illegal clock structure for GCC, with a generated clock derived from a top-level clock that was not defined. See Derived Clock on Input , on page 900. |
| FF-derived clock conversion disabled | FF-derived clocks have been detected, but clock conversion is not enabled. |
| Gating structure creates improper gating logic | Self-explanatory. Analyze and fix logic. |
| Illegal instance on clock path | Self-explanatory. Analyze and fix logic. |
| Incompatible clocks on mux inputs. | There is an illegal clock structure for GCC, with a generated clock derived from a top-level clock that was not defined. See Incompatible Clocks on Mux Inputs , on page 901. |
| Inferred clock from port | Self-explanatory notification. See Inferred Clock from Port , on page 901 for an example. |

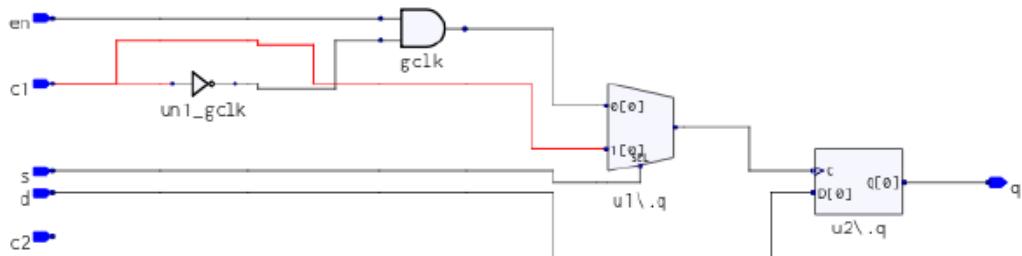
| Error Message | Explanation |
|--|--|
| Input clock depends on output | Self-explanatory. |
| Instance clocked by multiple variants of clock | The clock tree contains two phases of a clock. See Instance Clocked by Multiple Variants of Clock , on page 902. |
| Latch gated by OR originally on clock tree | Self-explanatory. |
| Multiple clock inputs on sequential instance | Self-explanatory. Analyze and fix logic. |
| Multiple clocks on generating sequential element | Self-explanatory. Analyze and fix logic. |
| Multiple clocks on instance | Multiple clocks found feeding an instance of a clock tree. See Multiple Clocks on Instance , on page 903. |
| Need declared clock or clock from port to derive clock from ff | Self-explanatory. Specify the clock. |
| No clocks found on inputs | No clocks found feeding an instance of a clock tree. |
| No generated or derived clock directive on output of sequential instance | Self-explanatory. Specify the clock |
| No hierarchical driver | Self-explanatory. Specify the driver. |
| Signal from port | Self-explanatory. |
| Unconverted clock gate | Self-explanatory. |
| Unable to determine clock driver on net | Self-explanatory. Specify the clock driver. |
| Unable to determine clock input on sequential instance | Self-explanatory. Analyze and fix logic. |
| Unable to follow clock across hierarchy | Self-explanatory. Analyze and fix logic |
| Unable to use latch as gated clock generator | Self-explanatory. Analyze and fix logic. |

Clock from Clock Mux

This message occurs when a mux is fed by a top-level clock on one input and a generated clock (derived from same top-level clock) on the second input, as shown below:



Another case that generates this message is when a mux is fed by a clock on one input and an inverted clock on the second input.

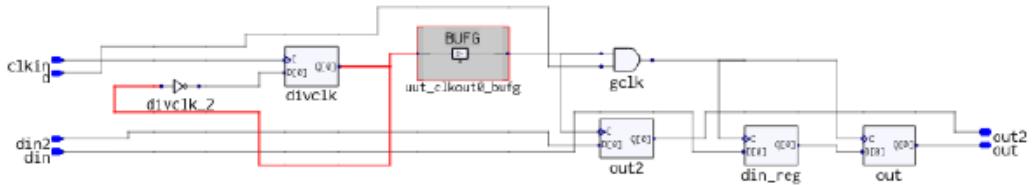


Action

One workaround is to create a faster clock using a generated clock (divide by 2), to replace the top-level clock. This clock structure will be supported in the futures releases.

Clock from Gated/Generated/Unconstrained Technology Primitive

This message warns that there is a gated/generated clock upstream from the technology clock primitive.



Action

This scenario can result in clock skew during place and route. Check for hold time violations on this path in the Vivado tool.

Clock from Generated Clock Directive

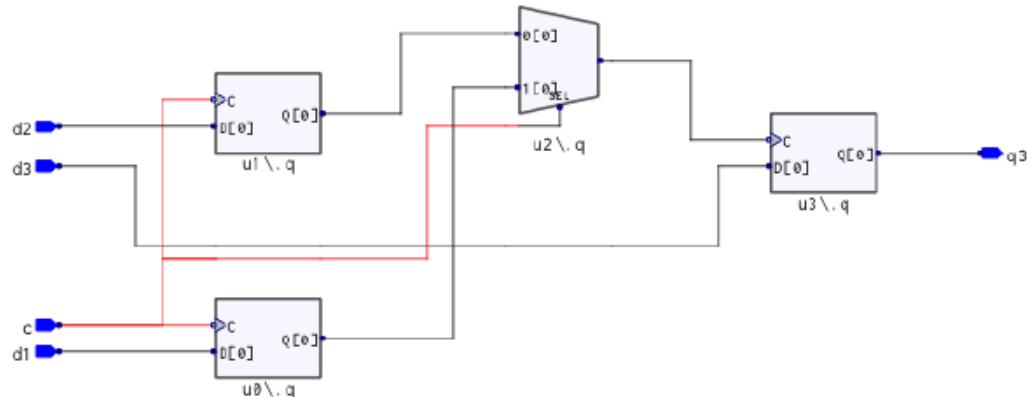
This message indicates that there is a `create_generated_clock` constraint on a mux output because the mux inputs are incompatible.

Action

This message about the constraint is informational. Address the root cause as described in [Incompatible Clocks on Mux Inputs, on page 901](#).

Clock on Mux Select Line

This message indicates an unsupported clock structure where a clock signal drives a select pin on a mux. When a select pin of a mux is driven by a clock, clock conversion stops at the output of the mux.

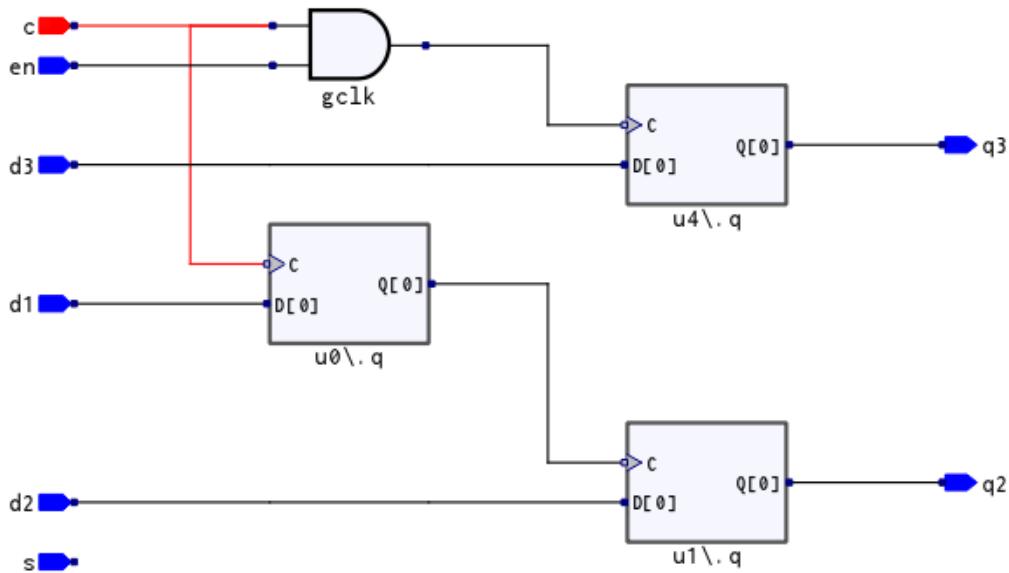


Action

Check the compile netlist to determine whether this structure is present in the RTL. If it is not, check that it is not a GCC tool error.

Derived Clock on Input

This message indicates a clock structure that is not legal for GCC: when the top-level clock is not defined and a generated clock is derived from it. See the example below, where the top-level clock `c` is not defined.

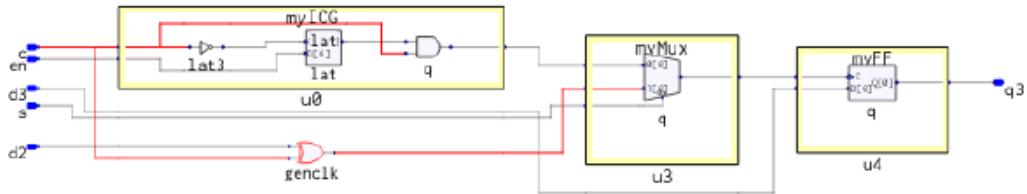


Action

Start with the sample instance (`u1` in the following image) which is reported as failed and trace its clock pin back to clock generator (`u0`). Then trace back the clock pin of clock generator to the undefined top level port `c`. In the FDC file, constrain the top-level port clock with a `create_clock` constraint, and constrain the derived clock with `create_generated_clock`.

Incompatible Clocks on Mux Inputs

This message is reported when the cone of logic driving one input leg of the mux cannot be optimized. The example below illustrates this by showing an e XOR gate, which is not a supported structure of GCC, driving an input leg of a mux.

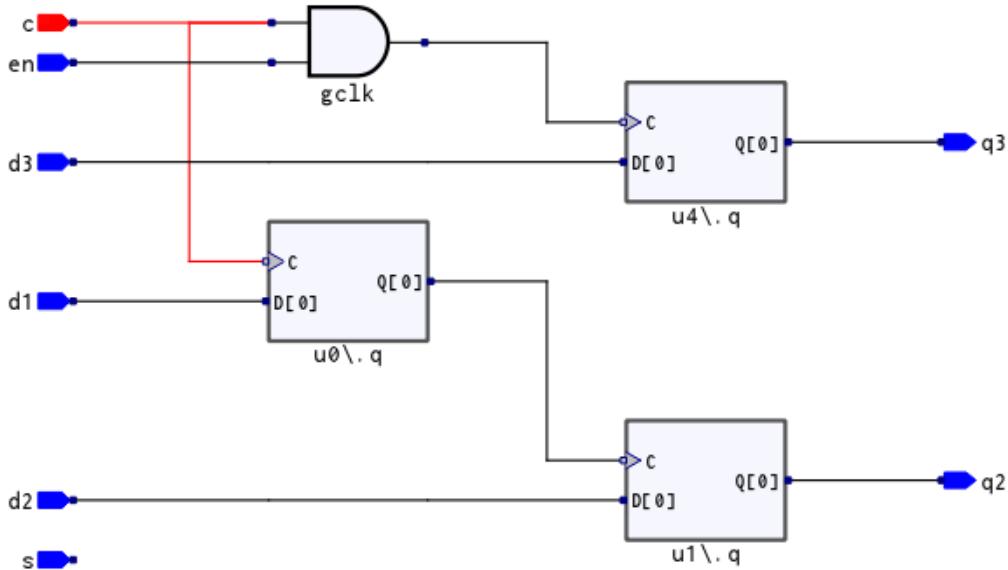


Action

Analyze the cone of logic driving each leg of mux for any unsupported clock structures.

Inferred Clock from Port

This message is reported when the top-level clock is not defined. The following example shows the instance u4 driven by a top-level clock, c, which is not defined.

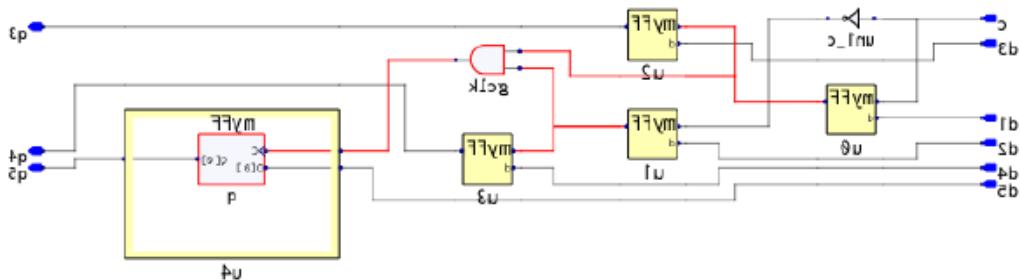


Action

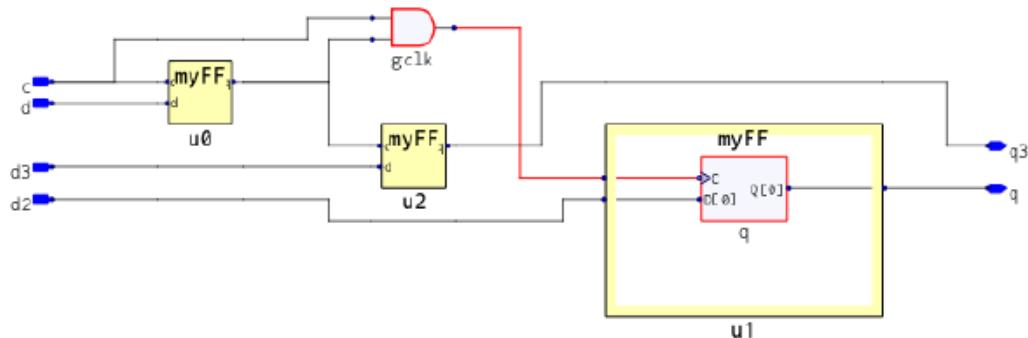
Trace the clock pin back and check the top-level port to make sure it is defined. You can also use the Clock Summary report to also locate the undefined clock. Apply the `create_clock` constraint on the top-level port clock in the FDC file.

Instance Clocked by Multiple Variants of Clock

This message is reported when a clock tree contains two phases of a clock. In the example given below, generated clocks from different phases of clock c are gated, and drive the clock pin of a sequential load.



In another scenario, clock c and the derived clock from c are both gated, and drive the clock pin of a sequential load.

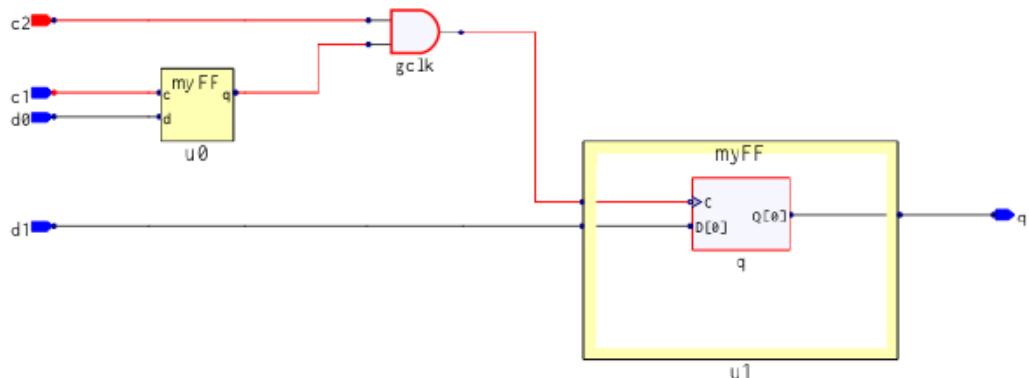


Action

If possible, tie off other test clocks, so that a single source clock feeds the clock tree.

Multiple Clocks on Instance

This message indicates that multiple unique clocks are being fed into a gated clock tree. Such a clock structure is not supported. In the example below, clock c2 and the generated clock from c1 are both gated, and they drive the clock pin of a sequential load.



There are two cases where a net is considered a generated clock: if a register that generates a clock directly drives a clock pin of a sequential element, or if a `create_generated_clock` constraint is defined for the registered output of a generating clock.

Action

If possible, tie off other test clocks, so that only a single source clock feeds the clock tree, or generate all clocks from a single-source clock.

Conversion of Datapaths with Latches

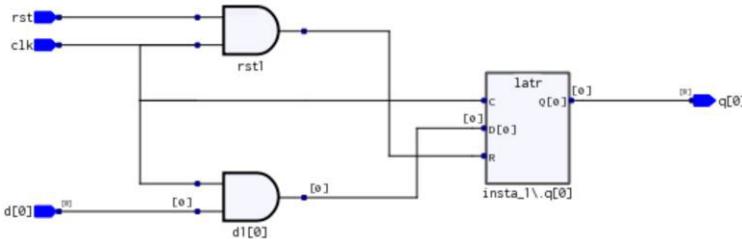
Latch inputs like data, set, and reset are sometimes dependent on latch enables, which could cause hold violations during place and route. When the `remove_clock_from_data` option is 1, it enables datapath conversion by removing the dependency of latch inputs on latch enable.

```
option set remove_clock_from_data 1
```

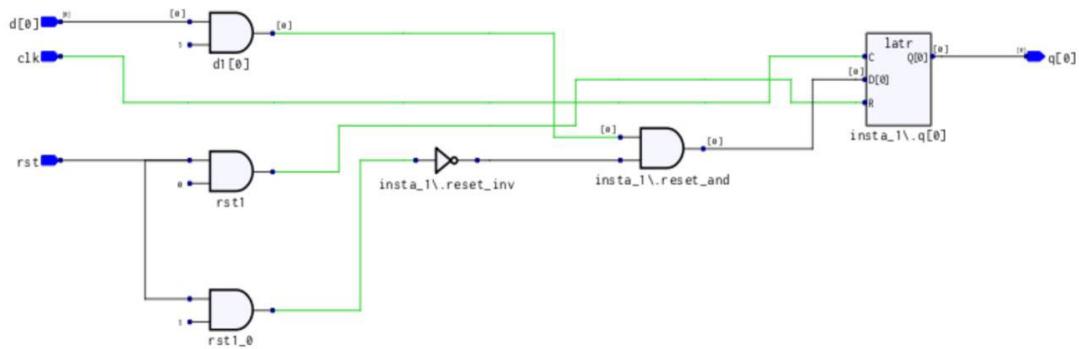
The default for this option is 1 (enabled). When a datapath is converted, the tool writes out a note in the `pre-map.srr` log file. In the note shown, `inst:insta_1.q_s[0]` is the latch instance and `D[0]` is the pin where the dependency was eliminated.

```
@N: BZ100 :"slowfs/thoma/test/latch.sv":16:0:16:0|Removing clock dependency of input pin:D[0] inst:insta_1.q_s[0] of PrimLib.latrs(prim) in view test
```

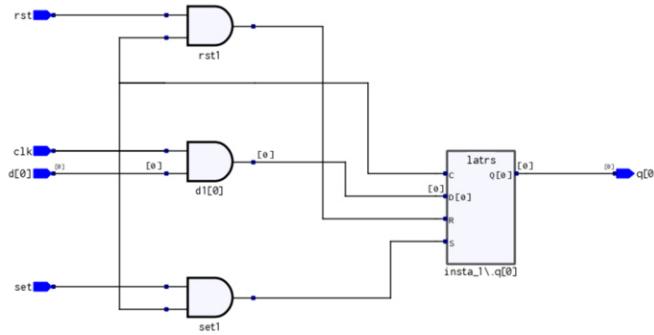
Example 1: Latch with Data, Reset, and Latch Enable



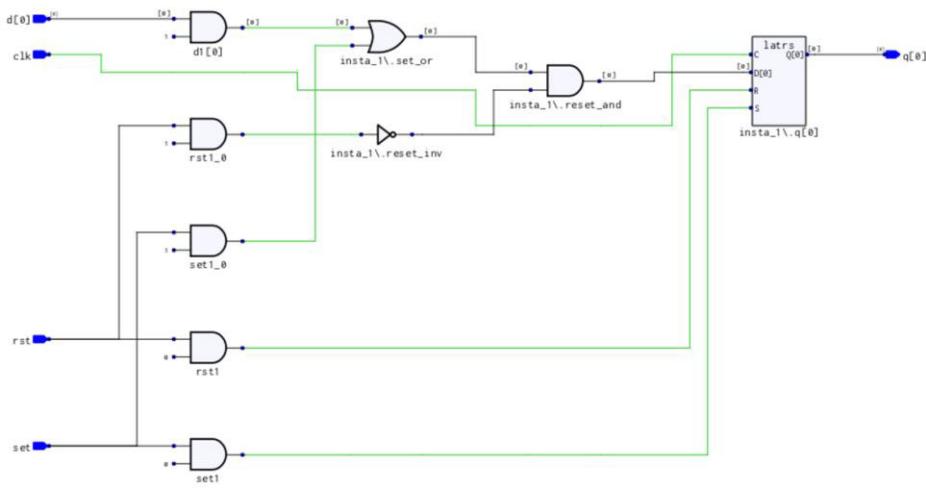
After conversion, the latch data and reset are independent of the latch enable:



Example 2: Latch with Data, Set, Reset and Latch Enable



After conversion, the latch data, set, and reset are independent of the latch enable.



Editing Netlists

Netlist editing offers a way to account for the architectural differences between ASICs and FPGAs, and make the ASIC netlist suitable for FPGA design. For information about netlist editing techniques, see these topics:

- [Editing Netlists with HAPS ProtoCompiler Commands](#), on page 907
- [Editing a Verilog Netlist](#), on page 910
- [Using Cross-Module Referencing \(XMR\)](#), on page 913
- [Using Gate-Level ASIC Netlists](#), on page 911
- [Creating Parallel Code with force and bind Statements](#), on page 911

Editing Netlists with HAPS ProtoCompiler Commands

One way to manage necessary variations between the ASIC and FPGA source files is to edit the ASIC netlist and modify it for FPGA synthesis. The prototyping tool provides netlist editing commands with the required functionality to do this.

Use this method:

- to replace ASIC-specific cells like clock-gating cells, make incremental changes to individual modules, insert buffers on high-fanout nets, and so on.
- to pre-process the design and stub named instances or all instances of a named view. You can specify a mix of views and named instances to be removed.

1. Compile the design.

You must start with a compiled database.

2. Create a Tcl script file that contains the changes you want to make to the netlist.

- Use the netlist editing commands to do this. See [Netlist Edit Command Reference](#), on page 343 in the *Command Reference Manual* for the syntax for the commands, and [Sample Netlist Editing Tcl File](#), on page 908 for an example of this file.

- If you want to run synthesis optimizations, add the optimization commands to the end of the netlist editing Tcl script so that the optimizations are performed on the edited netlist.
3. To apply the netlist edits, add the netlist editing file in compile command:

```
run compile -nle myNetlistScript.tcl
```

The tool applies the changes specified in the netlist editing file to the compiled database, and generates a new database state that reflects the design changes. It preserves the original netlist.

Note that if you have multiple instances of a module, the netlist editor applies the changes to the last instance in the RTL. This is because the netlist editor uses the compiler module name (which is the same for both instances) instead of the premap names, which are unique. Take `i_ilfsr_new` and `i_ilfsr`, which are two instances of the `ilfsr` module. If you use netlist editing to add a port to `i_ilfsr_new`, the tool inserts the port on `i_ilfsr`, because that is the name of the module in the netlist after compilation.

Note that, for design stubbing, if you use two commands on the same object with two different ties specified, the tool applies the first command. In the following example, `-tie 0` and `-tie 1` are specified in the netlist for instance `i1`. The first command `-tie 0` will be used for stubbing.

```
stub_inst {i1} -tie 0
stub_inst {i1} -tie 1
stub_view {add} -tie 0
```

Sample Netlist Editing Tcl File

The following example adds an RTL (srs) IP block and/or library primitives to the RTL database (srs) for a project.

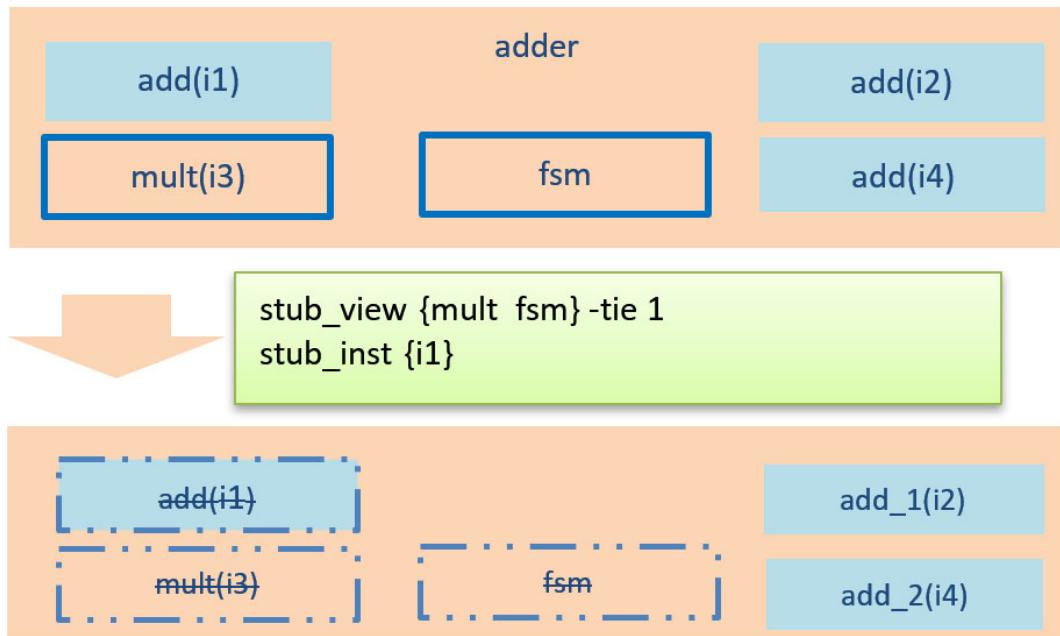
```
#Set the module to edit
define_current_view top
create_port {out[0:3]} -direction out
create_net {out[0:3]}
create_instance inst_A nle2_add_select
connect_net {out[0:3]} {inst_A.out[0:3]} {p:out[0:3]}
connect_net {d1[0:3]} {inst_A.a[0:3]}
connect_net {d2[0:3]} {inst_A.b[0:3]}
connect_net {d3[0:3]} {inst_A.c[0:3]}
```

```
connect_net {d4[0:3]} {inst_A.d[0:3]}
connect_net {d5[0:3]} {inst_A.e[0:3]}
connect_net clk1 inst_A.clk
connect_net sel inst_A.s0
connect_net reset inst_A.s1
#Insert an inv pair from the technology library (unisim.v)
#on the net driven by top_mult.out2[0]
insert_buffer -inverter_pair {top_mult.out2[0]} INV
```

Sample Netlist Editing Tcl File for Design Stubbing

The following example stubs instance i1, and views fsm and multi from the design:

```
#Set the module to edit
define_current_view adder
stub_view {fsm mult} -tie 1
stub_inst {i1}
```



Netlist Editing Objects

Netlist editing commands can be applied to collections and Tcl lists as well as individual objects. All arguments used with the netlist editing commands must apply to an existing object in the design, or to a constant. Identify objects with the object name or hierarchical path name.

Additionally, prefix a name with a qualifier to further define the object type. The following table lists the valid objects and their corresponding qualifiers:

| Valid Object | Object Name Syntax with Qualifier Prefix |
|--------------------|--|
| Pin connector | t:instanceName.pinName |
| Bit port connector | p:portName (top-level port) p:instanceName.portName |
| Net | n:netName n:instanceName.netName |
| Instance | i:instName i:instPath.instName |
| View | v:viewName |
| Library file | |
| Data-base file | |

In some commands, like single-argument commands for example, you can leave out the object qualifiers. In multi-argument commands, the arguments must be separated by spaces.

Editing a Verilog Netlist

Use the following method to edit a Verilog netlist:

1. Load the Verilog database.
2. Load the appropriate *.syn file for the target vendor from the lib/xilinx directory.

This file is required because the tool cannot work on the Verilog database directly. Use the views from the xilinx.syn file to create instances.

3. Create and add a Tcl file with the netlist editing commands to the synthesis project, as described in [Editing Netlists, on page 907](#).

See [Sample Netlist Editing Tcl File, on page 908](#) for an example of this file.

4. Synthesize the design as described in [Editing Netlists, on page 907](#).

Using Gate-Level ASIC Netlists

If you do not have the RTL for the ASIC design you must work with a gate-level netlist. This approach achieves lower performance than using the RTL. It uses more FPGA resources because the resources are mapped at a lower, cell level, and it does not map to high-level resources like RAM and DSPs. The advantage is that you do not have to modify the ASIC RTL.

1. Synthesize the ASIC design and generate a gate-level netlist.
2. Edit the netlist and replace the original ASIC cell library with the FPGA functional equivalents.
3. Use this netlist to run FPGA synthesis.

Creating Parallel Code with force and bind Statements

Use Verilog force and bind statements to separate the golden ASIC code from the code for FPGA prototyping. For example, you can use this method to substitute circuitry, change a clock tree, or stub out part of an ASIC design that is not needed in the FPGA prototype.

You can also use force and bind to and instances to debug in a non-invasive way. You can substitute a module interface definition to isolate it for debugging, or force a constant value to debug using a JTAG port, or create a custom debug port.

force Replaces existing drivers of internal signals in the hierarchy with new drivers. This statement is only supported for HAPS hardware targets.

bind Inserts an instance into the design hierarchy (for debugging).

1. Create a new file outside the ASIC module.

Alternatively, you can use the original RTL.

2. Define the alternative logic with the force and bind statements.
 - Define an initial block with the force construct, to override the ASIC-specific golden code in the original file.
 - Add the bind construct to insert the parallel instance that will replace the ASIC original.

See the example below for details.

3. In the prototyping software, specify this command before running pre-map:

```
option set bindandforce 1
```

During pre-mapping, the tool uses the forced values instead of the originals. If you do not specify this option, the tool ignores the bind and force statements you added.

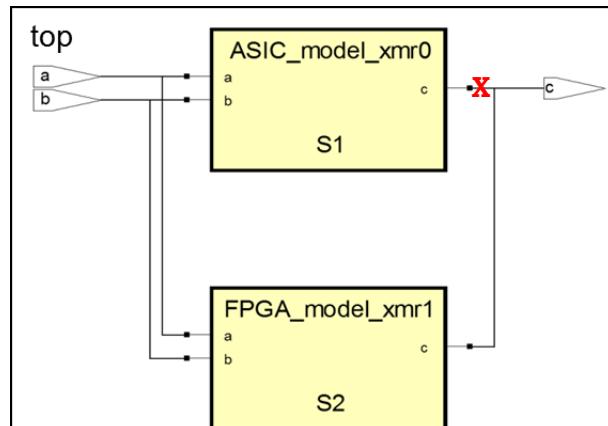
Example: Overriding Original Signal Driver

A typical use is to force a logic zero or one to scan flops to remove them from the prototype. In this example, file2.v illustrates how a new driver is inserted to override the original:

```
-----file1.v-----
//Golden code which can not be touched
module top(input a, b, output c);
  ASIC_model S1(a, b, c);
endmodule
module ASIC_model(input a, b, output c);
  //ASIC Specific Code
...
endmodule

-----file2.v-----
module FPGA_model(input a, b, output c);
initial begin
  force top.S1.c = top.S2.c;
end
//FPGA Specific Code
endmodule

bind top FPGA_model S2(a, b, c);
```



Using Cross-Module Referencing (XMR)

Cross-module referencing (XMR) is a hierarchical reference which provides a way to access signals that would otherwise be internal, without having to bring the signal up to the top level. It is typically used for verification applications, or to thread nets across modules for time-domain multiplexing (TDM). Cross-module references are also called out-of-module references (OOMR) or XREFs.

XMRs affect distributed processing operations, so you might not see a large runtime advantage in designs with a lot of XMRs.

There are three ways to specify cross-module references:

Define it in a cdc file [Cross-Module Referencing Using a CDC Constraint , on page 913](#).

Define it in the HDL [Defining Cross-Module References in the HDL , on page 914](#)

Use hyper connect modules [Using Hyper Source Modules for XMR , on page 915](#)

Cross-Module Referencing Using a CDC Constraint

The cdc constraint method is recommended, but you cannot use it in mixed VHDL and Verilog designs, because the compiler does not support cross-module referencing across languages.

1. Specify all the XMRs in a cdc file, using syn_connect_hrefs constraints, as in this example:

```
syn_connect_hrefs -readers {top.cnt_top} -writer {top.cnt_inst.cnt}
```

See [syn_connect_hrefs, on page 547](#) in the *Compiler Mapper Guide* for details of the syntax.

You can use cross-module references with compile points. In the partitioning flow, the tool creates black boxes after compilation for the connection elements (syn_hypersource and syn_hyperconnect), and this can be viewed in the schematic views. The actual connections are made after the pre-partitioning stage, just as with other nets.

Cross-module referencing has certain limitations:

- No cross-module referencing through an array of instances.
- No cross-module referencing into generate blocks.
- For upward cross-module referencing, the reference must be an absolute path from the top-level module.
- Do not use cross-module reference notation to access functions and tasks.
- You can only use cross-module referencing with Verilog/SystemVerilog elements. You cannot access VHDL elements with hierarchical references.

Defining Cross-Module References in the HDL

The recommended method to define XMRs is to use a constraint ([Cross-Module Referencing Using a CDC Constraint, on page 913](#)) but you can also mark XMRs directly in the HDL, across languages like Verilog and SystemVerilog. Note that you cannot use this method for mixed Verilog and VHDL designs, because the compiler does not support cross-module referencing across languages.

1. Add the connections manually to the HDL code, using periods (.) as the hierarchical operator for both upward and downward hierarchy in the XMRs, for any SystemVerilog and Verilog data types.
2. To specify a downward XMR read operation, use assign statements as shown here:

```
//Downward XMR Read
module top ( input a, input b, output c, output d );
    sub inst1 (.a(a), .b(b), .c(c) );
        assign d = inst1.a;
    endmodule
module sub ( input a, input b, output c );
    assign c = a & b;
endmodule
```

3. To specify an upward XMR write operation, use assign statements as shown here:

```
//Upward XMR Write
module top ( input a, input b, output c, output d );
    sub inst1 (.a(a), .b(b), .c(c) );
endmodule
module sub (input a, input b, output c );
    assign c = a & b;
    assign top.d = a;
endmodule
```

Using Hyper Source Modules for XMR

The recommended method to define XMRs is to use a constraint ([Cross-Module Referencing Using a CDC Constraint, on page 913](#)), but you can also use hyperconnect modules. You can use hyperconnect modules to define XMRs for mixed VHDL/Verilog designs, but use them with caution.

Threading Signals with Hyper Source Modules for Prototyping and IP

For prototyping, use Hyper Source to efficiently thread nets across multiple modules to the top-level design to support Time Domain Multiplexing (TDM).

You can also use it to easily replace an ASIC RAM with an FPGA RAM. Follow these guidelines to replace an ASIC RAM with an FPGA RAM:

1. Change the HDL for the RAM instantiation.
2. Add an extra clock signal to all the module interfaces.

Hyper source reduces the number of modified HDL modules to two, one for the RAM and one for the top level.

Using Hyper Source for IP Designs

For IP designs, Hyper Source is useful for validating and debugging the HDL without directly modifying it. After the HDL has been fully tested with complete QoR results, use Hyper Source to debug, as described in the following cases:

- Add some instrumentation logic that is not part of the original design, such as a cache profiler that counts cache misses or bus monitor that might count statistics about bus contention. The cache or bus might be buried deep inside the HDL; accessing the cache or the bus means ports might need to be added through several levels of hierarchy in the HDL.

The instrumentation logic can be included anywhere in the design, so you can use hyper source and hyper connect to easily thread the necessary connections during synthesis.

- Insert other hyper sourcing inside the IP to probe, monitor, and verify correct operation of known signals within the IP.

Threading Signals Through the Design Hierarchy of an IP

Use this procedure to thread a signal through the design hierarchy of a user IP and define an XMR. This signal can be threaded to a top-level port or signal even if the IP source is compiled separately. The tool automatically adds ports and signals between the source and the connection. Otherwise, these connections must be manually added to the HDL code.

However, if you use hyper source to thread signals through an EDIF, you must specify the Synthesis Strategy advanced mode.

The following procedure describes a method for using hyper source, using the example HDL shown in [Hyper Source Example, on page 917](#).

1. Instantiate the XMR source, using `syn_hyper_source`.
 - Signal `syn_hyper_source (in1)` module defines the source, with a width of 1.
 - The tag name "tag_name" is the global name for the hyper source.
2. Define how to access the hyper source which drives the local signal or port. The following apply to this example:
 - Signal `syn_hyper_connect (out1)` module defines the connection. The signal width of 1 matches the source.
 - Tag name can be the global name or the instance path to the hyper source.
3. In this hierarchical design, note the following about hyper source:
 - Applies to the module `lower_module`.
 - Signal `syn_hyper_source my_source(din)` module is defined for the source with a width of 8.
 - The tag name of "probe_sig" must match the name used in the hyper connect block to thread the signal properly.
4. In this hierarchical design, note the following about the hyper connect:

- Applies to the top-level module `top`, but can be any level of hierarchy.
- Signal `syn_hyper_connect connect_block (probe)` module is defined for the connection with a width of 8.
- Tag name of "probe_sig" must match the name used in the hyper source block to thread the signal properly.

5. After you run synthesis, the following message appears in the log file:

```
Available hyper_sources - for debug and ip models
HyperSrc label sub2_module.sub1_module.lower_module.probe_sig

Making connections to hyper_source modules
@N: : hyper_example.v(54) | Connected syn_hyper_connect hstdm_training_done_connect, label probe_sig
Finished RTL optimizations (Time elapsed 0h:00m:01s; Memory used current: 122MB peak: 129MB)
```

Hyper Source Example

```
/* Connect to a signal you want to export example : in1*/
module syn_hyper_source(in1) /*synthesis syn_black_box=1 syn_noprune=1 */;
parameter w = 1;
parameter tag = "tag_name"; /* global name of hyper_source */
input [w-1:0] in1;
endmodule

/* Use to access hyper_source and drive a local signal or port example
:out1 */
module syn_hyper_connect(out1) /* synthesis syn_black_box=1 syn_noprune=1 */;
parameter w = 1; /* width must match source */
parameter tag = "tag_name"; /* global name or instance path to hyper_source */
parameter dflt = 0;
parameter mustconnect = 1'b1;
output [w-1:0] out1;
endmodule

/* Example hierarchical design which uses hyper_source */
module lower_module (clk, dout, din1, din2, we);
output reg [7:0] dout;
input clk, we;
input [7:0] din1, din2;
wire [7:0] din;

syn_hyper_source my_source(din);
defparam my_source.tag = "probe_sig"; /* to thread the signal this
```

```
tag_name must match to name used in the hyper connect block */
defparam my_source.w = 8;

always @ (posedge clk)
if (we)
    dout <= din;
assign din = din1 & din2;
endmodule

module sub1_module (clk, dout, din1, din2, we);
output [7:0] dout;
input clk, we;
input [7:0] din1, din2;
lower_module lower_module (clk, dout, din1, din2, we);
endmodule

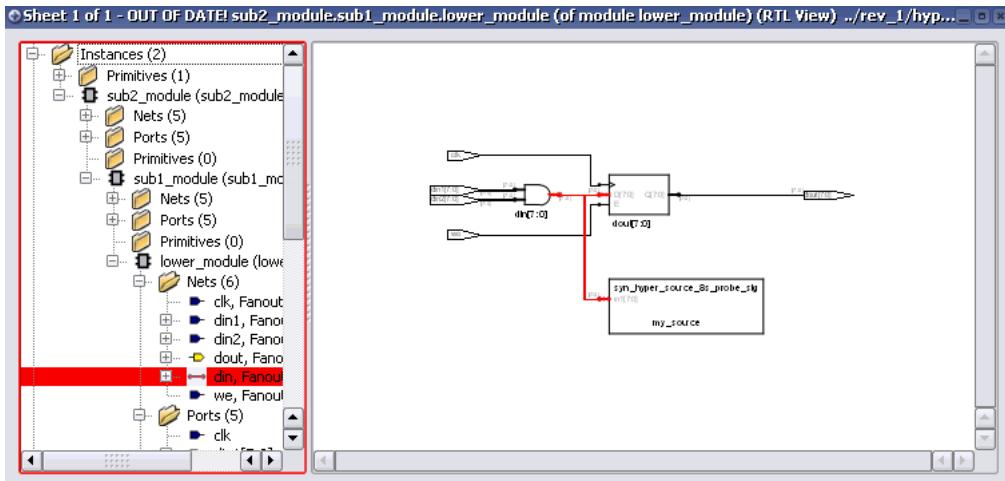
module sub2_module (clk, dout, din1, din2, we);
output [7:0] dout;
input clk, we;
input [7:0] din1, din2;
sub1_module sub1_module (clk, dout, din1, din2, we);
endmodule

module top (clk, dout, din1, din2, we, probe);
output [7:0] dout;
output [7:0] probe;
input clk, we;
input [7:0] din1, din2;

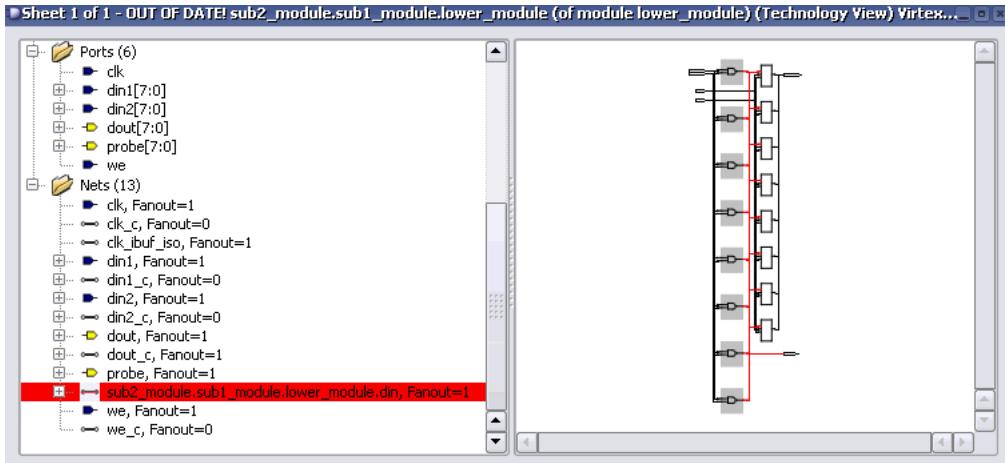
syn_hyper_connect connect_block(probe);
defparam connect_block.tag = "probe_sig"; /* to thread the signal this
tag_name must match to name used in the hyper connect block */
defparam connect_block.w = 8;

sub2_module sub2_module (clk, dout, din1, din2, we);
endmodule
```

RTL View



Technology View



Converting Designs for Unified Compile

The following topics describe differences between the native compiler and the unified compiler and how to convert designs from the native compiler.

- [Converting Older Designs for Unified Compiler](#), on page 920
- [Handling HDL Differences](#), on page 921

Converting Older Designs for Unified Compiler

To reuse a design that was compiled with the native compiler, use the `export` command, as described in the following procedure.

1. If need be, take care of differences in HDL implementation between the two compilers.

Follow the additional guidelines described in [Handling HDL Differences](#), on page 921.

2. Load the design database compiled with the native compiler, and export it using the `export uc` command.

```
export uc -path path_to_UC_dir
```

This generates the `UTF` file and the `VCS` script to run UC. The `UTF` file contains the following command:

```
vcs_exec_command {vcs_cmd.csh}
```

You can also specify the `export uc` command with `-zFmCheck`:

```
export uc -path <value> -zFmCheck
```

The corresponding `UTF` file contains an additional command to run `zFmCheck`:

```
vcs_exec_command { vcs_cmd.csh}
synthesis -generate_db_for_fmcheck true
```

3. Check the design file library.

This is a list of exported design files for a mixed language design where all the HDL is compiled in one library. The files include the files needed to run UC:

```
work_cmd_vlog.f
work_cmd_vhdl.f
synopsys_sim_setup
vcs_cmd.csh
uchaps.utf
uchaps.tcl
uc.prj
```

The `work_cmd_vlog.f` and `work_cmd_vhdl.f` files respectively specify that all Verilog files and VHDL files must be compiled in the work library. Use the `uc.prj` file as a reference to re-arrange the files in libraries or to create your own UC flow script.

4. Use the generated UTF file to launch unified compile from the synthesis tool.

```
launch uc -utf run.utf -ucdb <ucdb_dir>
```

5. Compile the ucdb generated by the unified compiler.

```
run compile -ucdb <ucdb_dir>
```

Continue with the rest of the flow as usual.

Known Limitations

- This process does not work with designs with configurations.
- The standard compiler implicitly adds the RTL file directory as part of the include path. This is not LRM-compliant and is not supported by unified compile.
- Multi-file compilation unit only works within a single library in VCS. If you need functions/tasks and definitions from different libraries, then the files defining them must be explicitly compiled in each library.

Handling HDL Differences

Generally, the conversion procedure described in [Converting Designs for Unified Compile , on page 920](#) is sufficient for most designs, but some designs might

need additional manipulation. Some of these scenarios are listed alphabetically below: **Compile Units**

There is a major difference between the native compiler in the prototyping tool and the UC compiler. The native compiler considers the whole design as one unit with Multiple File Compile Units (MFCUs). In the UC flow, each `vlogan` command is a separate compile unit, and you cannot have multiple file compile units across libraries. If a function or a macro is needed in a different library, it must be defined for that library.

To ensure that the compilation units are handled in the same way across both tools, first define a `dut.f` file that contains the RTL, include files, and definitions. Then add a single `vlogan` command like the one shown below to your script:

```
vlogan -full64 -work work -f dut.f
```

Compilation into Different Libraries

During design analysis, the VCS software stores the intermediate files in a design library or a logical library that points to a physical library. This physical library is a physical directory in your UNIX file system.

For details, refer to the VCS documentation.

1. Verify that the logical library is mapped to a physical directory in the `synopsys_sim.setup` file.
2. Specify this mapping in the `synopsys_sim.setup` file as shown in the example below:

```
WORK > DEFAULT
DEFAULT: ./worklib
```

In this example, WORK is the default logical library, which is now mapped to the physical library `worklib` in the UNIX file system. With this mapping, the VCS software stores all the intermediate files in the logical library WORK by default, and it generates an error if the logical library does not exist in the specified path.

Configurations in UC

Configurations are sets of rules that specify which source code to use in particular cases, as for enabling or disabling clauses; or the libraries from where to pick up modules. In the UC flow, systemverilog configurations must be defined with respect to the top-level module.

Configuration File Syntax

```
config config_identifier;
design [library_identifier.]cell_identifier;
config_rule_statement;
endconfig
```

| | |
|------------------------------|--|
| design | Keyword that starts a design statement for specifying the top level of the design |
| config_rule_statement | One or more of the following clauses: <ul style="list-style-type: none"> • default - Specifies logical libraries to search to resolve a default cell instance • instance - Specifies details about a specific instance. These details depend on the liblist. |
| liblist | Specifies the logical libraries to search to resolve all instances of the cell. |
| use | Specifies the cell definition is in the specified library |

You can specify the configurations in a top-level configuration file or in the RTL files.

Configurations in a Top-Level Configuration File

This is an example of a configuration set up in a single top-level configuration file, in `vcs_cmd.csh`: Don't see a top-level config file specified here--is it specified by `-top`?

```
vcs -top L1.topcfg -hw_top=top
vlogan rtl/lowl/lw.v -work Lw1
vlogan rtl/low2/lw.v -work Lw2
vlogan rtl/sub2/gt.v -work L2
vlogan rtl/sub3/gt.v -work L3
vlogan rtl/top.sv -work L1
vlogan rtl/top.v -work L1
```

This is the corresponding `synopsys_sim.setup` file. It specifies that module `lw.v` from the `rtl/lw1` folder be compiled into the library `Lw1`, module `lw.v` from the `rtl/lw2` folder be compiled into the library `Lw2`, and so on.

```
WORK > DEFAULT
DEFAULT : ./work
L1 : ./lib/lib1
L2 : ./lib/lib2
L3 : ./lib/lib3
Lw1 : ./lib/lw1
Lw2 : ./lib/lw2
```

Need clarification. What is the relationship between the config set up in the top-level config file (first code snippet above) and what follows below? If the stuff below is meant to be the contents of a top-level config file, where is that file specified? I don't see a top-level config file defined in the `vcs_cmd.csch` commands up top

The following example is the configuration definition. The library from which the instance is selected is based on the configuration. With the following configuration, the instance `top` is selected from library `L1`.

```
config topcfg; //Name of the config
design L1.top; //Location of the top-level module

default liblist L3 L2 Lw2; //Default search order for finding
instantiated modules
instance top.a2 liblist L2; //Explicitly naming library for the
following module

instance top.a3.i_lw liblist Lw1;
instance top.a2.g1 liblist Lw2; //Default chosen from Lw2 if not
defined below

endconfig
```

In the example above, the `top.a2` instance is selected from library `L2.gt`, the `top.a2.g1` instance is selected from library `LW2.lw` and so on.

```
library --> L1.top instance --> top --> this is @ top level
library --> L2.gt rtl instance --> top.a2 --> this is @
sublevel
library --> LW2.lw rtl instance --> top.a2.g1 --> this is @
lower level
library --> L3.gt rtl instance --> top.a3 --> this is @
sublevel
library --> LW1.lw rtl instance --> top.a3.i_lw --> this is @
lower level
```

Is this what you mean to say? Can I use this table instead?

| Library | Instance | Name | Hierarchical Level |
|---------|--------------|-------------|--------------------|
| L1.top | Instance | top | At top level |
| L2.gt | RTL instance | top.a2 | At sub-level |
| LW2.lw | RTL instance | top.a2.g1 | At lower level |
| L3.gt | RTL instance | top.a3 | At sub-level |
| LW1.lw | RTL instance | top.a3.i_lw | At lower level |

Configurations in RTL Files

This is an example of a configuration set up in a single top-level RTL file.

```
config in top.sv  rtl file
config topcfg;
    design L1.top;
    instance top.a2 use L2.gtconfig:config;
    instance top.a3 use L3.gtconfig:config;
endconfig

config in the gt.v  rtl file
config gtconfig;
    design L2.gt;
    instance gt.gl liblist Lw1;
endconfig
```

The `synopsys_sim.setup` file is set up as follows:

```
WORK > DEFAULT
DEFAULT : ./work
L1 : ./lib/lib1
L2 : ./lib/lib2
L3 : ./lib/lib3
Lw1 : ./lib/lw1
Lw2 : ./lib/lw2
```

The module `lw.v` from the `rtl/lw1` folder is compiled into the library `Lw1`, the module `lw.v` from the `rtl/lw2` folder is compiled into the library `Lw2`, and so on.

The library from which the instance is selected is based on the configuration. As a result of the configuration, the instance `top` is selected from the library `L1`.

```

library --> L1.top instance --> top    -- > this is @ top level
library --> L2.gt  rtl instance --> top.a2    -- > this is @
sublevel
library --> LW2.lw  rtl instance --> top.a2.g1   -- > this is @
lower level
library --> L3.gt  rtl instance --> top.a3    -- > this is @
sublevel
library --> LW1.lw  rtl instance --> top.a3.i_lw   -- > this is @
lower level

```

Here, based on the configurations in an individual file, the top.a2 instance is selected from the library L2.gt, the top.a2.g1 instance is selected from the library LW2.lw and so on.

Cross-Module Referencing

Cross-module references (XMRs) are hierarchical references that provide a way to access signals that would otherwise be internal, without having to bring the signal up to the top level.

Use this procedure to create an XMR for any SystemVerilog and Verilog data type:

1. Specify the XMRs in the RTL, not the .cdc file.

The UC flow does not support XMRs defined through directives in the .cdc file. Add the connections manually to the HDL code, using a period (.) as the hierarchical separator.

2. To specify a downward XMR read operation, use the `assign` statement as shown:

```

module top ( input a, input b, output c, output d );
    sub inst1 (.a(a), .b(b), .c(c) );
        assign d = inst1.a;
    endmodule

module sub ( input a, input b, output c );
    assign c = a & b;
endmodule

```

3. To specify an upward XMR write operation, use the `assign` statement as shown:

```

module top ( input a, input b, output c, output d );
    sub inst1 (.a(a), .b(b), .c(c) );
        assign c = a & b;
    endmodule

```

```
module sub (input a, input b, output c );
    assign c = a & b;
    assign top.d = a;
endmodule
```

Limitations

- XMRs defined through CDC constraints are not supported.
- In standard compile, XMRs cannot be used in mixed Verilog and VHDL designs, because the compiler does not support cross-module referencing across languages. However, the UC flow supports mixed language XMRs.

```
module top ( input a, input b, output c, output d , output e );
    sub inst1 (.a(a), .b(b), .c(c) );
    assign d = inst1.c1; // XMR Read
endmodule

entity sub is
    port ( a, b : in std_logic ;
           c : out std_logic);
end sub;

architecture sub_arch of sub is
    signal c1, x2 : std_logic;
begin
    c1 <= a or b;
    c <= c1;
end sub_arch;
```

Designs with VHDL

If the design contains VHDL, check the following:

- It is recommended that FDC constraints on VHDL paths be converted to upper case. You can also use the `-nocase` switch as shown below, but this method incurs a compile time penalty and all Verilog names will also be matched without considering case.

```
create_clock -name {clk_in1} -period 10 [get_pins
{top.vhdl_subopt.clk_in1}]
-nocase
```

- The VCS software does not support VHDL at the top-level. If the design has a top-level VHDL module, create a Verilog wrapper around the top-level VHDL.

Designs with Macros

Explicitly add required Verilog and VHDL macros in the VCS script.

Dynamic Force Statements

Dynamic force is like the force statement (see [Force and Release Statements, on page 930](#)), except that the force value is set dynamically.

1. In the RTL, use \$dumpvars syntax to specify the signals to be forced.

This is the syntax:

```
initial
begin : IICE_DF
  (* haps_force *) $dumpvars (1, <hier_path/signall_scalar>);
  (* haps_force *) $dumpvars (1, <hier_path	signal2_vector>);
  (* haps_monitor *) $dumpvars (1, <hier_path/signal3>);
end
```

Example:

```
initial
begin : IICE_DF
  (* haps_force *) $dumpvars (1, cnt_top.u_cnt_inst4.capture_cnt[15:0]);
begin...
```

2. In the prototyping software, enable options for \$dumpvars and force.

- Enable the debug_dumpvars and incremental_debug options.

```
option set debug_dumpvars 1
option set incremental_debug 1
```

- Optionally, enable global driver tracing and the insertion of a force mux on the real driver of the specified net by enabling the dynamic_force_global_driver option.

```
option set dynamic_force_global_driver 1
```

3. If the specified signals for force are not to be instrumented, define IICE with -mode none.

For example:

```
iice new {IICE_DF} -type regular -mode {none}
```

4. Run through the rest of the synthesis and place-and-route stages.

- At runtime, force the signal or perform readback: To force a dynamic value for a signal, specify the `force` command, where the format for the value can be decimal, hexadecimal, or binary ('d | 'h | 'b). The default format is decimal.

```
force {<hier_path/signal1_scalar>} ['d | 'h | 'b]<force_value>
```

The command can also be included in a script:

```
package require DF
# Open hardware for forcing and provide export folder for runtime
DF::df_open <export_result_folder> <HW_handle>

force {<hier_path/signal1_scalar>} <force_value>
force {<hier_path/signal2_vector>} [<format>]<force_value> # value
{<hier_path/signal1_scalar>}
# value {<hier_path/signal2_vector>}
# value {<hier_path/signal3>}
```

- For readback, optionally reset the default value format; then specify the value for force.

The default format for the read value is hexbinary (h), but you can set it to binary (b) using this syntax in the confrpsh shell:

```
DF::df_open <export_result_folder> <HW_handle> [h | b]
```

It can also be included in a script, following this syntax:

```
package require DF
# To open hardware for forcing and provide export folder for runtime
DF::df_open <export_result_folder> <HW_handle> [h | b]

# force {<hier_path/signal1_scalar>} <force_value>
# force {<hier_path/signal2_vector>} [<format>]<force_value>
# value {<hier_path/signal1_scalar>}
# value {<hier_path/signal2_vector>}
# value {<hier_path/signal3>}
```

Example of runtime script:

```
set hw_handle [list E025669 E025670]
DF::df_open ./export_results $hw_handle

force {cnt_top.u_cnt_inst4.capture_cnt[15:0]} 16'h00AA
value {cnt_top.u_cnt_inst4.capture_cnt[15:0]}
```

Force and Release Statements

The SystemVerilog force and release statements are procedural continuous assignments that allow expressions to be driven continuously onto variables or nets. There are differences in support for the force and release statements.

Force Statement

Both UC and standard compile flows support force statements. For example:

```
module top (
    input  a, b,
    output c, c1
);
asic_m s1 (a, b, c);
asic_m s2 (a, b, c1);
initial begin
    force s1.a = 1'b1 ;
end
endmodule
```

In standard compile, the force statement s1.a applies to the local net only, so it is only applies to instance s1. In UC, force applies to the global driver net, so the statement is applied to the a in the top module, so both instances s1 and s2 are affected by this force statement. Further, in standard compile, the force statement can only be specified in an initial block.

Release Statement

In standard compile, the release statement is ignored; in UC, the release statement is applied.

For example:

```
module top (
    input  a, b,
    output c, c1
);
asic_m s1 (a, b, c);
asic_m s2 (a, b, c1);
initial begin
    force s1.a = 1'b1 ;
    release s1.a ;
end
```

```
endmodule
```

Library Name Mapping

The VCS software allows you to create multiple logical libraries, each one pointing to a different physical library. The syntax to map a logical library to a physical library is shown below. Logical library names are case-insensitive.

```
logical_name: physical_name
```

The following example shows two logical libraries, ALU8 and ALU16, mapped to the alu_8bit and alu_16bit physical libraries, respectively. During design analysis, use the `-work` option to assign the files into the respective libraries.

```
vlog alu8.v      -work ALU8
vlog alu16.v     -work ALU16
```

In the run directory, specify the logical and physical memory mapping using the `synopsys_sim.setup` file.

```
WORK > DEFAULT
DEFAULT: ./work
```

Library Mapping

```
ALU8: ./alu_8bit
ALU16: ./alu_16bit
```

RTL Directory Paths

Check that the VCS script explicitly includes the RTL directory path. If it does not, it could result in an error:

```
Error-[SFCOR] Source file cannot be opened
Source file "test.h" cannot be opened for reading due to 'No such
file or directory'.
Please fix above issue and compile again.
"rtl/top.sv", 7
Source info: `include <test.h>
```

For the ProtoCompiler tool, you do not need to explicitly specify the RTL include directory because by default the tool searches the current HDL directory and then the include directories, sequentially.

However, this is not the case with the VCS software, which by default searches only the include directories. So, you must explicitly specify the RTL include directory in the `+includer` path for the UC flow. `+includer` specifies direc-

tories to search for include files used in the `include compiler directive. In the following example, `rtl` is the path of the RTL include directory, that contains the include files.

```
vlogan -sverilog +includer+rtl rtl/top.sv
```

Virtual Classes

You can use virtual classes to parameterize SystemVerilog static functions. UC supports a subset of virtual or abstract class features. The following constructs are supported:

- Parameterized abstract classes explicitly declared with the keyword `virtual`.

```
virtual class MyClass #(parameter SIZE=4, parameter SIZE1=1);
  typedef struct {
    logic [SIZE-1:0] elem1;
    logic [SIZE1-1:0] elem2;
  } my_type;
```

- Virtual classes declared in packages or by using the `$unit`.
 - Imported from a package.

```
package myPkg;
  virtual class ClassB;
    static function logic [1:0] F2 (logic [1:0] a);
    ...
    endfunction
  endclass
  endpackage

import myPkg::*;
module test (a,b,y,y1);
  ...
  assign y = ClassB::F2(b);
endmodule
```

- Referenced using the package scope, the class scope `::` operator

```
package pkg1;
  virtual class MyClass;
    static function int F1 ();
    ...
  endclass
endpackage
```

```

    endfunction
endclass
endpackage

module test (x,clk,y);
    ...
y = x + pkg1::MyClass::F1();
endmodule

```

- All legal specialization syntax. When all parameters have defaults, VC and VC#().

```
assign y3 =  pkg1::MyClassPkg#(.SIZE(S))::F2(x2);
y1.elem1 = MyClass#()::F1(x.elem1);
```

- Access to the following elements of a specialization using the class scope :: operator and access using a user-defined type name.

- Static functions with a default automatic lifetime

```

virtual class ClassA # (parameter SIZE = 32);
    static function automatic integer factorial (input [SIZE-1:0]
        operand);
    ...
endfunction: factorial
endclass

```

- User-defined type names (e.g a struct type)

```

virtual class MyClass #(parameter SIZE=4, parameter type
    mtype=logic );
    typedef struct {
        logic [SIZE-1:0] elem1;
        mtype           elem2;
    } my_type;
endclass

```

- Parameters and local parameters

- Non-parameterized abstract classes explicitly declared with the keyword virtual.

```

virtual class MyClass;
    static function int F1();
    ...
endfunction
endclass

```

- Inherited using the `extends` keyword.

```

virtual class MyClass #(parameter SIZE=4 );
    static function logic [SIZE-1:0] F1 (logic [SIZE-1:0] x);
        endfunction
endclass

class MyClass_ext extends MyClass;
    static function logic [SIZE-1:0] F1 (logic [SIZE-1:0] x);
        return ~x;
    endfunction
endclass

```

- Nested declarations (using a class declaration as a class item)

```

virtual class ClassB # (parameter SIZE = 2);
    typedef struct {
        logic [SIZE-1:0] elem1;
        logic elem2;
    } my_type;

virtual class ClassA;
    static function logic [1:0] F2 (logic [1:0]x);
        return |x;
    endfunction
endclass

endclass

```

Limitations to Virtual Class Support

The following virtual class constructs are not supported for the UC flow.

- Abstract classes without an explicit `virtual` keyword
- Static data properties
- Static tasks
- Variables of an abstract class type
- Static functions with static lifetime or containing variables with a static lifetime

- Forward class declarations
- Prototypes
- The implements keyword

CHAPTER 8

Verifying the Design

This chapter describes how to formally verify your design using the Formality equivalence checker. You run formal verification after you complete synthesis. See the following for the details:

- [Guidelines for Successful Formal Verification](#), on page 938
- [Verifying Single-FPGA Designs](#), on page 945
- [Verifying Multi-FPGA Designs](#), on page 953
- [Setting Options to Guide Formal Verification](#), on page 958
- [Running VCS Simulation for Inferred Memories](#), on page 960
- [Running Formal Verification with Checkpoints](#), on page 962

For other methods of verifying your design at different points in the design cycle, see these topics:

- [Running Simulation](#), on page 557
- [Ensuring Hardware Bring-up](#), on page 621
- [Instrumenting and Debugging Designs](#), on page 633

Guidelines for Successful Formal Verification

See the following subjects for information about best practices when using the Formality equivalence checker to verify your design:

- [General Guidelines for Checking Your Design](#), on page 938
- [General Coding Guidelines for Formal Verification](#), on page 939
- [Coding Guidelines for Verilog and SystemVerilog Designs](#), on page 940
- [Limitations to Equivalence Checking](#), on page 944

General Guidelines for Checking Your Design

The following tips provide some guidelines for successfully checking your design with the Formality tool:

- Generally, non-equivalent points are the result of missing or incorrect mapping rules. Evaluate any non-equivalent points carefully to determine the cause. In most cases, fixing annotations resolves the problem. Search for errors and warnings in the formal verification tool log file and resolve them.
- Although the final goal is a top-down verification run with no non-equivalent points, it is recommended that you begin verification at the sub-module level and work up to the top level. Small issues at the sub-module level can translate into a large number of unmapped or non-equivalent points at the top level. There is a smaller amount of design logic at the sub-module level, so it is much easier to deal with and resolve annotation issues at this level.
- When extracting a sub-module from an entire design for verification, preserve the sub-module interface as it is in the RTL. During synthesis, the tools optimizes across hierarchies and can alter module level interfaces. For module-level verification, you can either apply synthesis attributes to the module to preserve the interface, or synthesize the module by itself.

General Coding Guidelines for Formal Verification

The following coding practices are recommended to avoid potential mismatches in formal verification.

Formality Guidelines

- always_comb statements
 - Put these statements on lines by themselves; do not share the line with other statement.
 - Avoid using labels for always_comb statements, because it leads to a naming mismatch for the datapath operators that contain the statement:

```
always_comb
begin : label //naming mismatch for datapath operators
    result = a * b;
end
```
 - Avoid using always_comb statements inside loops.
- Variable sizes must match when assigned or mapped to one another to avoid any potential formal verification mismatch. Look for warnings that report a port map width mismatch or an index out of range, like CD285 and CD543, for example.
- Do not re-declare parameters and other variables within the same scope.
- Track loop iterations carefully. You can set a loop limit in the Formality setup Tcl script with `set hdlin_while_loop_iterations limit`.
- Always explicitly declare functions as automatic.
- Keep math operations of variables on the same line, and group them as two operands per operator, as shown:
$$(((a + b) + c) + d)$$
- Do not use force and bind.

VC Formal Guidelines

- Sizing of variables should match when assign or mapped to one another. Avoid out of bound access or assignments.
- Parameters and other variables should not be redeclared within the same scope.

- When using `include, do not give paths, rather use the include directives in the project settings.
- Avoid using escaped names.
- Automatic instantiation for block boxes is not supported.
- Latch inferencing.
 - Avoid writing conditional assignments for latches. For example: assign result = ctrl ? in : result;

The synthesis tool will infer a latch for the above assignment, whereas VC Formal may not infer any latch. This could lead to inconclusive or failing verification. Instead write the assignment in an always block as follows

```
always @ctrl
if (ctrl)
    result = in;
else
    result = result;
```

This will make both the tools infer latches leading to successful verification.

- Avoid usage of synthesis compiler design constraints (CDC) that cause design changes, such as "syn_append_submodules" for module renaming.
- Avoid usage of supply0/supply1 port types.
- Try to wrap memories into smaller modules, this will make the module to be partitioned for verification reducing runtime.

Coding Guidelines for Verilog and SystemVerilog Designs

Follow these Verilog and SystemVerilog coding guidelines to help the verification process run smoothly:

- Avoid using unions in SystemVerilog.
- When using 'include, do not use paths, but use include directives in the settings, instead.
- The synthesis engine ignores simulation-specific constructs like \$DISPLAY and \$SETUPHOLD but the Formality checker does not. Enclose these constructs in //synthesis translate_off/on pragmas.

- Always size data types. For example, use `1'b0` instead of `b0`.
- Do not use duplicate modules or entities in the same design. The synthesis engine supports the `allow multiple modules` option, which allows the last read definition to be selected when different instances of the same module have different contents. However this option causes ambiguity in the Formality checker and is not permitted. Make sure each design has only one definition of each module or entity.
- When using custom packages, provide the function definition for operators like `+` and `-`. Include the package definition file along with the other source files if you use a non-standard package. For example, if you use the non-standard library listed below, you must add the following file: `vhdl_src/std_developerskit/synthreg.vhd`.

```
library std_developerskit;
use std_developerskit.synth_regpak.all;
```

- Use `full_case` and `parallel_case` with caution. If you specify `full_case` and do not define all the cases, it can cause problems during verification because the Formality checker does not honor this directive. The hardware behavior is not affected.
- Use pragmas and synthesis directives with caution, because they direct the tool to remove logic based on these directives. As these directives are typically specified by the designer, it is assumed that the board behavior will be correct. However, simulators do not read these directives and do not do any reachability analysis.
- Do not use `force` and `bind` statements.
- Avoid using multiple drivers, where a net is driven both by a constant and an active signal. Synthesis resolves this to a constant, but the equivalence checker might not do the same. The Formality tool treats this as a wired AND.
- Gated and generated clock structures

By default, the synthesis flow converts gated and generated clock structures, to ensure that the prototype works on the board. You can help the verification process by identifying and constraining the correct clocks before synthesis.

- RTL RAMs
 - RAMs are typically implemented as block RAM in FPGAs, and these RAMs do not have good verification models. Take the RTL code from

which synthesis would infer a RAM and enclose it in a separate hierarchy. Put the RAM code in a separate design hierarchy with a fixed boundary (`syn_hier=“fixed”`). This allows the tool to black box the RAM and continue verifying of the rest of the design.

- The tool automatically black boxes all user models with inferred RAM. The RAMs are synthesized, but black-boxed for equivalence checking.
- If the code would infer register banks instead of a RAM implementation, use the `syn_ramstyle=“register”` attribute to force it to be implemented as registers, because registers can be formally verified.

- Multipliers

The tool does not black box multipliers. Small multiplier structures (<13 bits) that are implemented as logic can be verified formally. Currently, large multipliers mapped to logic could take a long time to verify and the results might still be inconclusive. For larger structures, you might want to enclose them in a separate hierarchy and verify them by other means like simulation.

- DSP blocks

Logic inferred as DSP blocks is typically difficult to verify because pipelined registers from surrounding logic go into the DSP block. Currently the tool verifies logic mapped to DSPs without registers, and automatically ensures that the mapped DSPs are combinational, unless you have instantiated them in the RTL.

For DSPs with registers, the current workaround is to direct the tool to implement the DSP blocks as logic using the `syn_DSPstyle` attribute, or to enclose the DSP structure in its own hierarchy so it can be black-boxed. Verify these black blocks by some other means like simulation. Future releases will include support for DSPs with registers.

- Retiming

Turn off retiming option whenever possible. Wherever possible, turn it off on specific blocks, not globally. When you enable Verification Mode, this optimization is automatically disabled.

- `syn_hier=“fixed”`

When synthesizing, it helps to use the `syn_hier=“fixed”` attribute on all levels of hierarchy. Add a simple collection command like the following to the `fdc` file:

```
define_scope_collection all_views {find -hier -view {*}}
define_attribute {$all_views} {syn_hier} {fixed}
```

- Registers with set and reset pins

Black-box any blocks containing RTL code that describes registers with both set and reset pins. The Xilinx tool does not have a single equivalent primitive that has both pins. It uses two registers, which causes problems for the Formality checker.

- Latches

Describe latches explicitly within `always` blocks, instead of assigning constants. Constant assignment is ambiguous, and different tools might interpret the assignments differently. Also, it might result in combinational loops, which you are advised to avoid.

Use the explicit style:

Explicit Coding Style

```
module latch_test (input ctrl, data, output reg result1, result2);
  always @ (data or ctrl)
    if (ctrl)
      result = data
endmodule
```

Ambiguous Coding Style

(Do not use)

```
module latch_test (input ctrl, data, output reg result1);
  assign result1 = ctrl ? data : result1
endmodule
```

- XMRs

You can use downward cross-module reference (XMR) constructs. This example shows a high-level port (`d`) that refers to a port at the lower level in `inst1`:

```
module top (input a, input b, output c, output d);
  sub inst1 (.a(a), .b(b), .c(c));
    assign d = inst1.a;
  endmodule

module sub (input a, input b, output c);
  assign c = a&b;
endmodule
```

The Formality software does not support the following:

- Upward references.

- References to the same level.
- References to packed structures.
- XMRs specified with relative paths. Use an absolute path instead: assign a = top.inst2.b.

Limitations to Equivalence Checking

There are some limitations to using both the Formality and VC Formal software:

- The tool automatically turns off the following synthesis optimizations.
 - Finite state machines inference
 - Retiming and pipelining
 - High-reliability synthesis

Additional limitations for the Formality software:

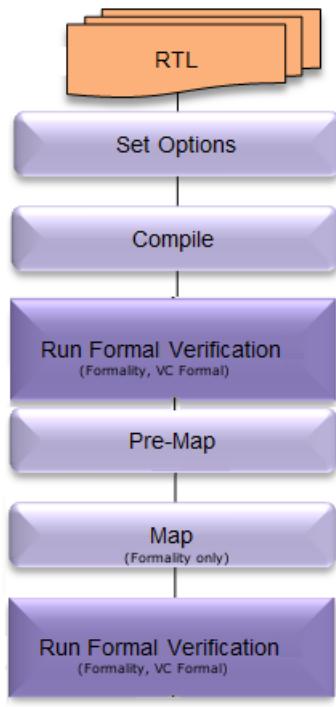
- Modules with inferred memories are automatically black-boxed. The tool reports the memories and provides the setup needed to run VCS simulation. See *Running VCS Simulation for Inferred Memories*, on page 960.
- VHDL support is limited. VHDL designs that use complex math operators might not get verified properly.
- For single-FPGA designs, verifies the mapped netlist against the RTL language description
- For multi-FPGA designs, verifies the multi-FPGA mapped netlist against the post-partitioning output.
- Verification mode automatically turns off some synthesis optimizations so that the design can be formally verified. Verification mode preserves hierarchy so as to converge in the formal verification flow. For designs with critical timing requirements, after verification, re-synthesize with verification mode turned off, so that the tool can optimize the design.

Additional limitations for the VC Formal software:

- System Verilog bind is not supported.

Verifying Single-FPGA Designs

For single-FPGA designs, you run formal verification (VC Formal™ or Formality) on a mapped design and compare the results to the original RTL. The figure shows the flow and the following table summarizes the commands in the flow:



The procedure below provides step-by-step details:

1. Follow the guidelines described in [General Coding Guidelines for Formal Verification, on page 939](#), and note the limitations ([Limitations to Equivalence Checking, on page 944](#)).
2. Set up the environment for formal verification:
 - To ensure that the tool does not run out of memory, set virtual memory and ulimit in the shell:

```
limit vmemoryuse unlimited
ulimit unlimited
```

- Set the environment variables listed below. If you use GUI mode, you are prompted to set the environment variables later, but if you run in shell mode, it is your responsibility to set the variables.

| Environment Variable | Description |
|----------------------|--|
| VC_STATIC_HOME | Path to the VC Formal executable; required for VC Formal verification |
| FORMALITY | Path to the Formality executable; required for Formality formal verification |
| VCS_HOME | Path to the VCS tool; required to simulate inferred memories or unverifiable blocks |
| XILINX_VIVADO/XILINX | Path to the Xilinx tools; required for formal verification libraries from the vendor |

- If you have DesignWare® components, you must also set the path to the DesignWare library; otherwise the compiler errors out.
Either set dc_root in the options.tcl file to point to the Design Compiler® installation (option set dc_root), or specify the path to the installation with the SYNOPSYS variable. If both \$SYNOPSYS and dc_root are specified, the tool uses dc_root. If only \$SYNOPSYS is defined, the tool uses that path.

3. Set verification options and synthesize the design.

- Enable formal verification by setting this command:

```
option set verification_mode 1
```

This command turns off synthesis optimizations that cannot be formally verified. In designs with critical timing requirements, the design as verified with Formality might fail on the board. For such designs, after verification, re-synthesize with verification mode turned off.

- If you want to do bottom-up verification or distribute the verification problem among smaller blocks, use the option set hier_verification command to define the number of verification blocks and do bottom-up verification. The default (auto) uses heuristics to automatically divide the design into verification blocks, but you can set it to manual, and then specify a percentage you want to use to create verification blocks with the option set hier_verification percent command. A percentage of 5 specifies that modules that are at least 5 percent of the total area

be made into verification blocks. If you set `hier_verification` to off, formal verification runs on the whole design. For large designs, it is recommended that you avoid the off setting, and use auto instead.

- Run synthesis. You can run verification after the compile (VC Formal or Formality) or map (Formality only) stages. The following table shows the sequence of commands required from each database state:

| From Compile State | From Map State |
|--|---|
| option set verification_mode 1 | option set verification_mode 1 |
| run compile | run compile run pre_map (Formality can include UPF files) run map |
| export verification (see step 4) launch vcf launch formality (see step 5) | export verification (see step 4) launch formality (see step 5) |

See [General Coding Guidelines for Formal Verification, on page 939](#) and [Coding Guidelines for Verilog and SystemVerilog Designs, on page 940](#) for information about the source files.

You can include UPF and UUM files for formal verification at the pre-map stage. For example:

```
run pre_map -upf test.upf
```

- When you verify post-compile Verilog against HDL for designs with memory blocks, the tool automatically verifies the memories without black-boxing them.

There are some limitations to this coverage of memories. The tool automatically black boxes some memories that it does not support: multi-process write, partial read and write and bit-wise read memories.

Verification fails if the design contains memories inferred from multiple register banks, negative index register banks, or a mixture of to and downto register banks; for example `reg [0:15] mem [4095:0]` or `reg [15:0] mem [0:4095]`.

You can encounter long formal verification runtimes because of the time taken for elaboration.

- Check for warnings that might cause mismatches during formal verification, then fix them. For example:

CD285

@W: Port map width mismatch (7 => 8) on port of component <addern>

CD453

@W: Index <8> may be out of range

4. Export the verification data with the `export verification -path` command.

The `export verification -path` command exports the synthesized data for verification. The directory includes encrypted svf guidance files that direct the verification run, based on the synthesis optimizations performed. At this point, the files are encrypted and you cannot modify them. You can export the files from a system generate, compiled, or mapped state.

It is strongly recommended that the files not be moved once exported. Note that the Tcl files in the exported directory require access to the source files. If you transfer the exported files to another location and the Tcl files cannot access the source files, the formal verification run will have problems.

5. Start the VC Formal or Formality tool from the command line or the GUI after synthesis is complete.

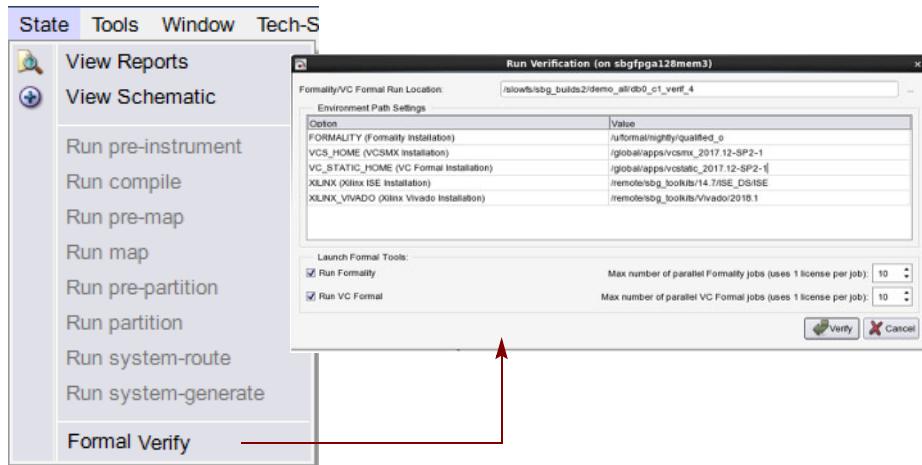
- Optionally, set parameters for the run. See [Setting Options to Guide Formal Verification, on page 958](#) for the options you can set.
- It is recommended that you launch the formal verification software from within the prototyping tool. To start the tool from the prototyping tool shell, type the command listed below. The optional `-gui` argument launches the formal verification software GUI mode. If you do not specify `-gui`, the tool launches it in shell mode and runs it in the background. If you specify `-gui`, it launches the formal verification software in GUI mode.

```
launch formality -path runDir -gui  
launch vcf -path runDir -gui
```

- To start the tool from the prototyping GUI, go to a mapped database state and select Formal Verify from the State popup menu.

In the dialog box that opens, specify the directory for the formal verification run. The default location is

`workingDir/databaseName_state_verif`. You can also define the environment variables if you did not do so earlier.



- Select either Run Formality or Run VC Formal or both.
- Click Verify. The formal verification tool starts equivalence checking in the specified directory and compares the implementation to the reference RTL. It creates the verification partitions and runs the formal verification tool on those. The verification partitions are run individually for better visibility and debugging.

The formal verification tool does not support CDPL for running these verification partitions in parallel, but you can distribute processing on multiple processors on the same machine.

If you run equivalence checking in GUI mode, a reporting window opens. The window is updated automatically as the run progresses.

- To view the results in the log file, use the `view report` command in the prototyping window.

6. Analyze the results in the interactive reporting window.

Formality Status

| Report View | | Formality Status | VCFormal Status |
|------------------------------------|---|------------------|---|
| Formality Information | | | |
| Open Formality GUI | | | Notes: 1. Set the appropriate FORMALITY environment variable before running verification 2. Runs RTL Vs Post-Compile netlist verification in the exported directory 3. Verification failure may not always be a logic issue 4. The generated scripts and files should not be modified |
| Path | /uiformms/ngrndly/cuasifd/m_shell | | |
| Run Directory | /slowts/stg_build2/same_alltbo_c1_verif_5post_compile | | |
| Guidelines | readme | | |
| Verification Matching Script | top_compile.match.tcl | | |
| Verification Black Boxes | top_compile.tb.sv | | |
| Compare Points Summary | compare_points_summary | | |
| Status | Finished in 19.00 seconds | | |

| VC Formal verification of modules containing inferred memories | |
|--|---|
| Modules | VC Formal Verification Notes: 1. Set the appropriate VC_STATIC_HOME environment variable before running VC Formal 2. Runs RTL Vs Post-Compile verification for modules containing inferred memories VC Formal verification script |
| Modules with inferred memories | (Note: Click once, may take some time to bring up the GUI) Run VC Formal (gui mode) |
| Detailed Verification Report | |

| Instantiation of modules containing inferred memories Formality verification of memories is not currently supported | | | | | |
|--|--------------|----------------|---|--|---|
| Total Ignored Compare Points : 261 | | | | | |
| Instance | Resynthesize | Compare Points | Compare Points (for inferred memories) | VCS Simulation | |
| top + memblock_inst | Run | 261 | 261 | VCS Info VCS Run Directory: (Run Directory)design_infermemblock/vcs_simulation Testbench: memblock_tb.v Stimulus: memblock.stm Run log: vcs.log Status: Not Run (Note: Click once, may take some time to bring up the GUI) Stimulus (VCS Run Directory)/memvcs | Simulate All (batch mode) |

| Post Compile Formality Verification Summary (Status : Verification SUCCEEDED) | | | | | | |
|---|---------------------------|--------------|--|--------------|-------------------------------------|---------|
| Design | Verification Black Box | Status | Compare Points passing vs. total (408 / 408) | Hint | Report | Session |
| top | more | Succeeded | 315 / 315 | --- | top_compile.log | --- |
| + pgm_ctrl | --- | Succeeded | 51 / 51 | --- | pgmctrl_compile.log | --- |
| + data_mux | --- | Succeeded | 32 / 32 | --- | datamux_compile.log | --- |
| + alu | --- | Inconclusive | 10 / 10 | Resynthesize | alu_compile.log | open |

[Detailed Verification Report](#)

VC Formal Status

The screenshot shows the 'VC Formal Status' interface. At the top, there are tabs: Report View, Formality Status, and VC Formal Status. The 'VC Formal Status' tab is active.

VC Formal Information:

| | |
|--------------------------|---|
| Open VC Formal GUI | Notes: 1. Set the environment variable VC_STATIC_HOME environment variable before running verification 2. Run RTL Vs Post-Compile netlist verification in the exported directory 3. Verification failure may not always be a logic issue 4. The generated scripts and files should not be modified |
| Path | /u/producting/mgny/patches/VCS2017.12_VCSTAT/Orchestrated/INSTALL/bin/vc |
| Run Directory | /slow/stg/builds/202003_000_01_VC11/post_compile |
| Guidelines | None |
| Verification Black Boxes | top_compile_vs1_sv |
| Assertions Summary | assertions_summary |
| Status | Finished In 2.75 minutes |

Post Compile VC Formal Verification Summary (Status : Verification SUCCEEDED):

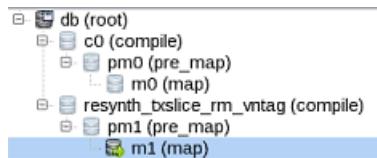
| Design | Verification Block Box | Status | Assertions passing vs. total (45 / 45) | Min | Report | Session |
|---------------|------------------------|-----------|---|-----|------------------------|---------|
| top | none | Succeeded | 37 / 37 | --- | top_compile.log | --- |
| -> pigrm_ctrl | — | Succeeded | 1 / 1 | --- | pigrm_ctrl_compile.log | --- |
| -> data_mux | — | Succeeded | 3 / 3 | --- | data_mux_compile.log | --- |
| -> ali | — | Succeeded | 3 / 3 | --- | ali_compile.log | --- |
| -> memblock | — | Succeeded | 1 / 1 | --- | memblock_compile.log | --- |

Detailed Verification Report

- Check the Verification Summary status, which updates in real time as the run progresses.
 - Check the Hint column for Resynthesize links. The tool generates these links for verification blocks that fail, and the link lets you resynthesize and verify it separately.
 - Check the Instantiation of Modules Containing Inferred Memories section. These are verification blocks with inferred memories, which cannot be formally verified with Formality. Instead, verify with VC Formal or use VCS to simulate these blocks. See *Running VCS Simulation for Inferred Memories*, on page 960 for the procedure.
 - Check the black box report for modules that are defined as synthesis black boxes. Modules with generated clocks and modules with RAMs are black-boxed automatically.
 - To view the interactive report in a browser, click Detailed Verification Report.
7. If the report shows failures, try these techniques to analyze and fix the design:
- Click the link in the Report column to view the log, which shows the points where the design failed.
 - To view the cone of logic, open the formal verification software run from the Session column in the report, and view the schematic for the failing logic.

- For failing verification blocks, try resynthesizing them.
- If your design fails because it is too complex, use option set `hier_verification_percent` to set a lower percentage, to reduce complexity and create smaller verification blocks.

Rerun compile, pre-map, and map. When you resynthesize, the tool does not overwrite the originals, but creates parallel database states:

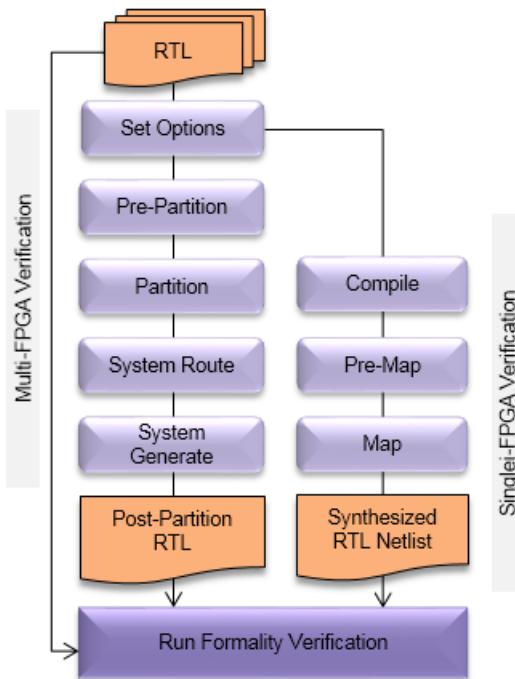


8. To run incremental verification, make adjustments as needed, and rerun the equivalence checker.
 - For information about the changes in guidance you can make in the public `svf` file and other options you can set, see [Setting Options to Guide Formal Verification, on page 958](#).
 - Do not change the `svf` directory structure or try and edit the encrypted `svf` files.
 - Rerun the equivalence checker.

The checker runs incrementally and only verifies the verification partitions that have modifications, so this run is faster.

Verifying Multi-FPGA Designs

For multi-FPGA designs, you verify the partitioned RTL against the original RTL. You can also verify the mapped RTL for each partitioned FPGA against the post-partitioning RTL netlist. The following figure summarizes the steps in the flow, and the table summarizes the commands in the flow.



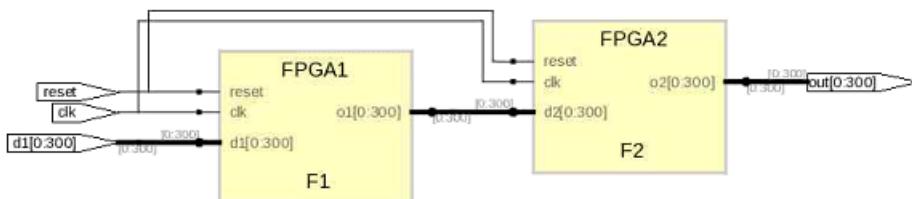
| Post-Compile | After System Generate | Post-map |
|---|--|---|
| option set verification_mode 1 | option set verification_mode 1 | option set verification_mode 1 |
| run compile | run compile run pre_partition run partition run system_route run system_generate | run compile run pre_map run map |
| export verification launch formality | export verification launch formality | export verification launch formality |

The following procedure provides step-by-step details.

1. Set up the design for verification.
 - Set memory limits.
 - Set environment variables for the Formality software, and other tools.
 - Set verification options in the prototyping tool.

For more information about these steps, see [Verifying Single-FPGA Designs, on page 945](#).

2. To run formal verification after compile, do the following:
 - Compile the design as usual (run compile).
 - Follow the directions in step 4 to export the database and launch Formality.
 - Analyze the design as described in steps 5-7.
3. To run formal verification after top-level partitioning, partition the design and generate partitions.
 - Make sure that design flow is set to partition (option set design_flow partition).
 - Use the usual sequence of commands to partition: run compile, run pre_partition, run partition, run system_route, and run system_generate.



- You can include UPF files for formal verification by specifying the files when you run pre-partition:


```
run pre_partition -upf test.upf -tss board.tcl
```
- 4. To run verification, after creating partitions with run system generate, do the following from the system generate state.
 - In shell mode, use the export verification and launch formality commands. If you use shell mode, make sure to specify the environment variables for the tools, as you will not be prompted for them.

- Click Verify in the GUI to start the Formality run and verify the post-partitioned netlist against the original RTL. If you run equivalence checking in GUI mode, a reporting window opens. The window is updated automatically as the run progresses. To view the results in the log file, use the view report command in the prototyping window.

Formality Information

| | |
|-------------------------------------|---|
| Open Formality GUI | <p>Notes:</p> <ol style="list-style-type: none"> Set the appropriate FORMALITY environment variable before running verification Run RTL Vs Synthesis netlist verification in the generated 'verif' directory Verification failure may not always be a logic issue The generated scripts and files should be carefully modified, ONLY if required |
| Path | /global/app/itm_2016.03-SP2/bin/tm_shell |
| DesignWare Path | /global/app/syn_2013.02-SP4 |
| Run Directory | /rambox/log/support/zaman/formality/Bfrt uc_lab/formal_verification/post_map |
| Guidelines | readme.txt |
| Verification Script | exec regis.map.tcl |
| Verification Matching Script | exec regis.map.match.tcl |
| Verification Black Boxes | exec regis.map.bb.tcl |
| Compare Points Summary | compare_points_summary |
| Status | Finished in 8.78 seconds |

Post Map Formality Verification Summary (status : Verification FAILED)

| Post Map Formality Verification Summary (status : Verification FAILED) 93.60% Design is Formality Verified (excludes verification black boxes) | | | | | | | |
|--|------------------------|------------------------|-----------|--|--------------|----------------------------------|----------------------|
| Design | Verification Black Box | Area (% of the design) | Status | Compare Points passing vs. total (339 / 359) | Hint | Report | Session |
| eight_bit uc | none | — | Succeeded | 323 / 323 | — | eightbit.map.log | — |
| data_mux | — | 16.40% | Failed | 15 / 92 | Resynthesize | datamux.map.log | open |

[Detailed Verification Report](#)

Post Map Formality Verification Summary (Status : Not Run)

| Design | Parameter | Verification Black Box | Area (% of the design) | Status | Hint | Report | Session |
|--------------|-----------|------------------------|------------------------|---------|------|----------------------------------|---------|
| eight_bit uc | — | none | — | Not Run | — | eightbit.map.log | — |
| alu | none | — | 10.84% | Not Run | — | alu.map.log | — |
| data_mux | — | — | 16.19% | Not Run | — | datamux.map.log | — |

[Detailed Verification Report](#)

5. Analyze the results in the interactive reporting window.

The figure below shows part of the reporting window, showing the verification partitions and a link to a detailed verification report.

| Post Map Formally Verification Summary (Status : Verification FAILED) | | | | | | | | |
|---|------------------------|------------------------|-----------|--|--------|--------------|-------------------|---------|
| Design | Verification Black Box | Area (% of the design) | Status | Compare Points passing vs. total (339 / 355) | | Hint | Report | Session |
| | | | | Passed | Failed | | | |
| eight_bit_uc | mos | — | Succeeded | 323 | 323 | --- | eightbituc.maplog | --- |
| data_mux | — | 16.40% | Failed | 15 | 32 | Resynthesize | datamux.maplog | open |
| Detailed Verification Report | | | | | | | | |

| Post Map Formality Verification Summary (Status : Not Run) | | | | | | | |
|--|-----------|------------------------|------------------------|---------|------|-------------------|---------|
| Design | Parameter | Verification Black Box | Area (% of the design) | Status | Hint | Report | Session |
| | | | | | | | |
| eight_bit_uc | — | mos | — | Not Run | --- | eightbituc.maplog | --- |
| alu | more | — | 10.84% | Not Run | --- | alu.maplog | --- |
| data_mux | — | — | 16.15% | Not Run | --- | datamux.maplog | --- |
| Detailed Verification Report | | | | | | | |

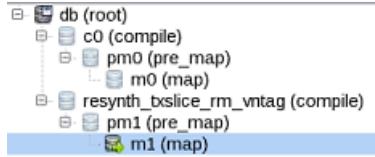
- To view the interactive report in a browser outside the tool, click [Detailed Verification Report](#).
- Check the Verification Summary status, which updates in real time as the run progresses.
- Check the Hint column for Resynthesize links. The tool generates these links for verification blocks that fail, and the link lets you resynthesize and verify it separately.
- Check the Instantiation of Modules Containing Inferred Memories section. These are verification blocks with inferred memories, which cannot be formally verified. Instead, use VCS to simulate these blocks. See [Running VCS Simulation for Inferred Memories](#), on page 960 for the procedure.
- Check the black box section of the report for modules that are defined as black boxes. This includes user-defined black boxes or HAPS interfaces. The tool checks the connectivity around the black boxes, but does not verify the insides of the black boxes.

| Verification Black Box Report | | |
|----------------------------------|-----------------------------|--|
| Total Ignored Compare Points : 0 | | |
| Instance | Compare Points | Note |
| eight_bit_uc + amibus_pllinst | umr_clk_gen_v7_eight_bit_uc | HAPS Interface Instance. Please verify separately. |
| eight_bit_uc + hdo_ip | hdpainit_v7_0_1a_16_08_1s | HAPS Interface Instance. Please verify separately. |

6. If the report shows failures, do the following:

- Click the link in the Report column to view the log, which shows the points where the design failed.

- To view the cone of logic, open the Formality run from the Session column in the report, and view the schematic for the failing logic.
- If your design fails because it is too complex, use option set `hier_verification_percent` to set a lower percentage to reduce complexity and create smaller verification blocks. When you resynthesize, the tool does not overwrite the originals, but creates parallel database states:



7. To run incremental verification, make adjustments as needed, and rerun the equivalence checker.
 - For information about the changes in guidance you can make in the public svf file and other options you can set, see [Setting Options to Guide Formal Verification, on page 958](#).
 - Do not change the svf directory structure or try and edit the encrypted svf files.
 - Rerun the equivalence checker.

The checker runs incrementally and only verifies the verification partitions that have modifications, so this run is faster.

Setting Options to Guide Formal Verification

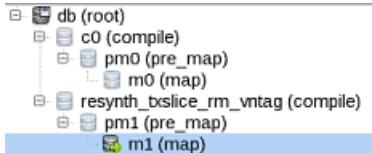
After analyzing the first run, you probably need to make some adjustments and rerun verification. The tool has options and guidance commands you can work with to influence verification.

1. To fine-tune the automatic verification partitions that the tool creates, specify the option set hier_verification manual command, and then use the option set hier_verification_percent command to set a percentage value for generating the verification blocks.

A percentage of 5 specifies that modules that are at least 5 percent of the total area be made into verification blocks. For complex designs that fail, set a lower percentage with option set verification_hier_percent to reduce complexity and create smaller verification blocks.

2. Rerun verification with the new option settings.

When you verify the design again, the tool does not overwrite the original database state, but creates a new parallel database state:



3. Edit formal verification guidance commands in the decrypted public svf file and then rerun verification.

The Setup Verification for Formality (.svf) script is a file that contains Formality commands. The svf commands provide guidance so that the Formality tool can account for the transformations made by the synthesis optimizations. For example, the tool might duplicate a register to improve drive strength, and this file records the register duplication.

- Do not try and edit the encrypted private svf file. The decrypted public svf file that is editable (`formality_svf/design_map.svf`) is generated after a Formality run.
- You can modify the following commands in the public svf file, but make sure you understand the effect your changes will have on the verification process:

| To... | Use These Commands ... |
|---------------------------------------|--|
| Modify the design | <code>guide_uniquify</code> <code>guide_ununiquify</code> <code>guide_ungroup</code> <code>guide_instance_merging</code> |
| Edit sequential optimization settings | <code>guide_reg_merging</code> <code>guide_reg_constant</code> <code>guide_reg_duplicationg</code> <code>uide_reg_removal</code> <code>guide_inv_push</code> |
| Guide port-matching and black-boxing | <code>set_user_match</code> <code>remove_user_match</code> <code>set_direction</code> <code>set_black_box</code> |

For details about these commands, refer to the Formality or VC Formal documentation.

- Do not edit any `guide_private` commands in the public svf file. These commands are placeholders for commands in the private svf and must not be altered.

Running VCS Simulation for Inferred Memories

The Formality tool cannot verify modules with inferred memories, user-defined black boxes, or HAPS interface IP, and treats them as black boxes. However, you can verify these black boxes by either using VC Formal or running VCS simulation on them as described below.

1. Make sure you have the VCS_HOME environment variable set to point to the VCS software.

You must use the VCS-MX software to run simulation.

2. Run synthesis and formal verification, as described in *Verifying Single-FPGA Designs*, on page 945 and *Verifying Multi-FPGA Designs*, on page 953.

The modules must be Verilog modules.

The tool automatically creates test patterns and a testbench for running VCS simulation on modules with inferred memories. The test pattern is randomly generated. The testbench is generated using the clock and reset information from the design partition that is created for running simulation. The information is listed in the interactive reporting window.

It also creates a VCS simulation directory called `verif`.

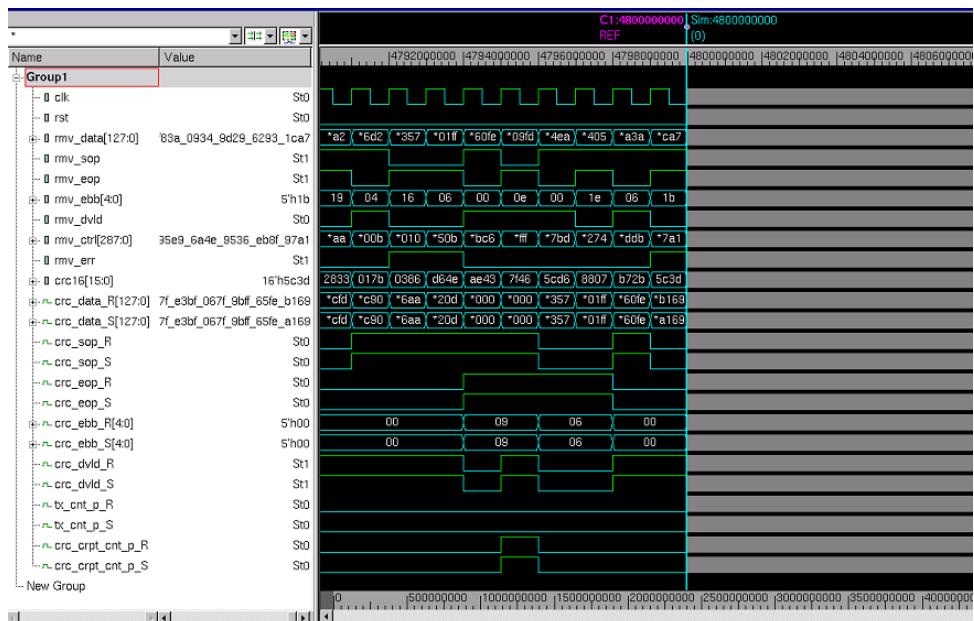
3. In the formal verification reporting window, do the following:

- Check the reported memories or IP:

| Installation of modules containing inferred memo des Formality verification of memories is not currently supported | | | | | |
|---|------------------|--|-------------------|--|--|
| Total Ignored Compare Points : 261 | | | | | |
| Instance | Résynthèse | Estimated Area (% of the design) | Compare Points | Compare Points (for inferred memories) | VCS Simulation |
| <code>eight_bit uc + ROM</code> | <code>PLI</code> | 5.9% | 0 | 0 | <p>VCS Info VCS Run Directory (Run Directory design_mif vcom vcs_simulation)</p> <p>Testbench: <code>inst_rom_tb</code> Stimulus: <code>inst_rom_stm</code> Run log: <code>vcs.log</code></p> <p>Status: Not Run</p> <p>Simulate VCS Run Directory/verif</p> |

- Check the files generated and edit the test pattern or stimulus as needed. Edit the testbench as needed.
4. Run simulation, using one of the methods below:
- From the report window, click the appropriate simulate link in the VCS Simulation column to run simulation for an individual block or for all the blocks. Only click the link once.
- Clicking starts the launch script and the VCS run. The tool compiles the individual module or all the synthesized modules as specified, and simulates the synthesized netlist against the original RTL. The VCS GUI opens.
5. Check the results in the VCS run log.

Use the GUI for further debugging.



Running Formal Verification with Checkpoints

This verification flow is based on the creation of check points, which can be compared to the original Verilog using the VC Formal tool to run formal verification.

1. Set environment variables for the verification tools:
 - Set the Formality installation (FORMALITY).
 - Set the VCS variable (VCS_HOME).
 - Set the VCS Static variable (VCS_STATIC_HOME).
 - Set the Xilinx Vivado installation (XILINX_VIVADO).
2. Set options from the GUI or the command line:
 - Enable verification mode: option set verification_mode 1
 - Enable the creation of checkpoints: option set verify_check_points 1. When this option is enabled, the tool automatically creates checkpoints based on the design modules (SCMs).
- After the compile stage, the tool writes out checkpoints for the modules. Each verification check point contains the following data that the verification tools can use:
 - Complete environment setup, formal tool setup, and launching data
 - Complete reference and implementation Verilog designs, and other data
 - Formal tool constraints such as clocks, black boxes etc.
 - Guidance and matching data for achieving maximum formal convergence
 - Interface for controlling various capabilities of specific formal tools based on the check point requirements
 - Report generation mechanisms
 - Handshaking mechanisms between the prototyping and verification tools
3. Export files for formal verification.
 - Go to a valid database state, such as compile or pre-map. For a list of the database states, refer to [export verification, on page 60](#).

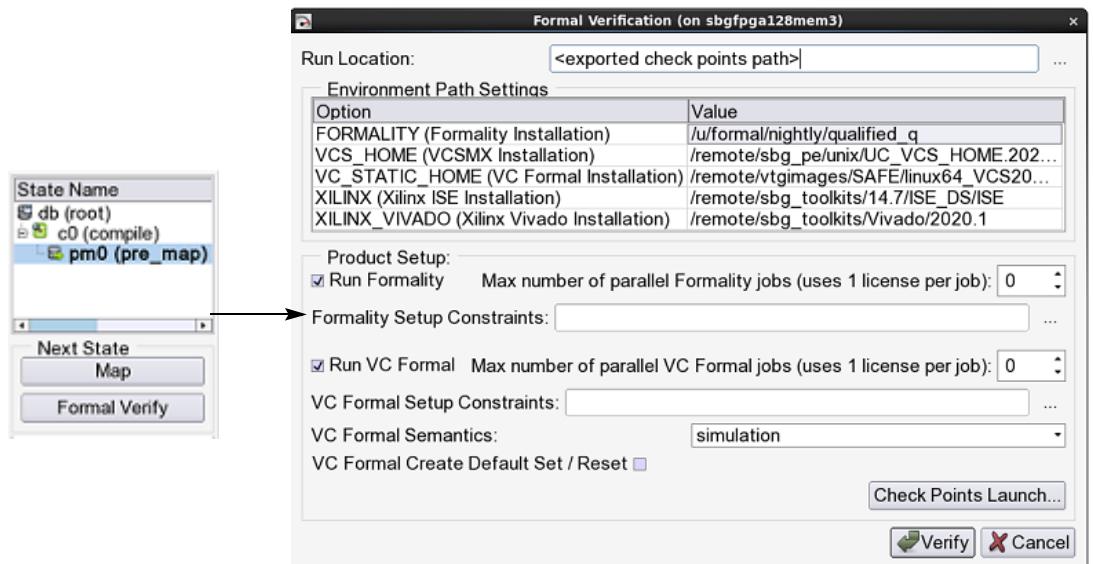
- Run the export verification command.

```
export verification -check_points -path dir
```

4. Specify verification tool options and launch the verification check from the command line or from the GUI.

The GUI provides an easy way to enter parameters for the verification check.

- To use the command line, specify the formal_verify command from a valid database state. For the syntax of this command, see [formal_verify, on page 62](#). You can also do this from the GUI. From the pre-map or other valid stage, click Formal Verify, and fill in the verification tool controls in the dialog box that opens.
- In the GUI or the command, specify controls for the verification tools, such as the maximum number of parallel jobs. You can also specify a setup constraints file for Formality or VC Formal with the verification control commands.
- Click Check Points Launch. This opens a window for the results.



5. Analyze results.

Example 1

Verification Check Points After Compile

```
database load db -autocreate -technology HAPS-80
option set verification_mode 1
option set verify_check_points 1
launch uc
run compile
export verification -check_points -path check_points_data
set fvId [formal_verify -path check_points_data -post_compile]
run pre_map
run map
job wait $fvId
```

Verification Check Points After Pre-Map

```
database load db -autocreate -technology HAPS-80
option set verification_mode 1
option set verify_check_points 1
launch uc
run compile
run pre_map
export verification -check_points -path check_points_data
set fvId [formal_verify -path check_points_data -post_premap]
run map
job wait $fvId
```

Example 2

```
database load pcucdb -autocreate -technology HAPS-80
option set verification_mode 1
option set verify_check_points 1
launch uc -utf run.utf -ucdb ucdb -v 2.0
run compile -ucdb ucdb
export verification -check_points -path check_points_data
set fvcId [formal_verify -path check_points_data -post_compile ]
job wait $fvcId
```

For more information, see [verification_mode](#) and [verify_check_points](#) options in [option, on page 81](#).

CHAPTER 9

Recommended Methodologies

This chapter describes recommended prototyping methodologies to achieve specific goals:

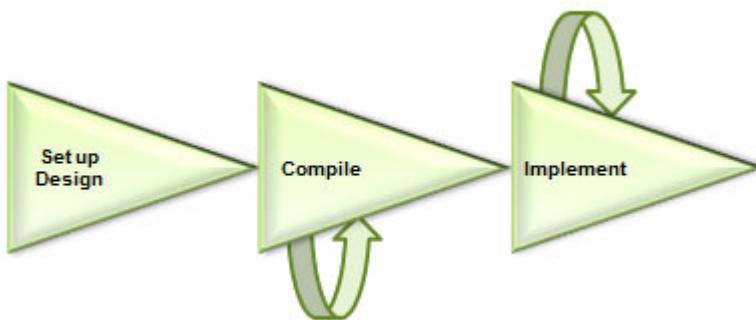
- [Methodology for a Quick Prototype](#), on page 966
- [Generating a Best-Performance Prototype](#), on page 973
- [Exploring Different Partitioning Solutions](#), on page 977
- [Scripting Design Tasks](#), on page 979

Methodology for a Quick Prototype

Typically, you need an initial prototype for proof-of-concept scenarios, like exploring FPGA architectures, developing prototypes, pipe-cleaning the design, or validating constraints. In this scenario, the focus is on quickly bringing up an initial FPGA prototype. The priorities are a smooth migration process from ASIC to FPGA, fast turnaround times through iterations, and quick board bring-up.

The HAPS prototyping methodology ensures fast bring-up with a three-pronged approach to reducing the overall time: fast implementation algorithms, incremental runs, and fewer iterations.

The figure shows the phases for implementing a quick prototype, and then describes the best practices for each phase in the sections below. Note that although setup and compilation are shown as two different phases to make the iterative process clear, in practice the automatic setup features are only implemented when you compile the design. The Implement step encompasses pre-mapping, mapping, placement, and routing.



- [Setting up the FPGA Design and Compiling It, on page 967](#)
- [Synthesizing, Placing, and Routing a Quick Prototype, on page 970](#)
- [Tips for Reducing Runtime, on page 972](#)

For a quick prototype most of the time is invested up front, in getting the design set up. As a quick prototype is typically the starting point for a more mature prototype focused on better performance, this initial setup time also serves to shorten the implementation time for a subsequent, performance-focused prototype.

Setting up the FPGA Design and Compiling It

The HAPS prototyping solution offers many features to automate or speed up the process of making ASIC RTL FPGA-friendly. Many setup features run automatically when you compile the design. This helps to compress the time frame. For details, refer to [Converting ASIC Designs to FPGA, on page 33](#) and [Creating and Compiling Databases, on page 57](#).

Setting up the Design

The steps to set up a quick prototype are the same, so the following procedure lists various aspects of the setup process, and the settings or features that you can use to shorten the time taken for design preparation.

- RTL support

The prototyping tool reads standard Verilog and VHDL, as well as newer versions like SystemVerilog; this ability eliminates the need to process the RTL.

- ASIC RTL netlist translation for FPGA design
 - There are three broad techniques to use when reconciling ASIC-FPGA RTL differences:

| | |
|------------|---|
| Conversion | Convert ASIC gated clocks with automatic gated clock conversion (run during compilation). Automatically convert generated clocks (run during compilation). For more information about converting ASIC clocking, see Converting ASIC Clocks , on page 42 . |
|------------|---|

| | |
|--------------|--|
| Substitution | Substitute ASIC memories like system RAM and ROM, system FLASH, system EEPROM, and memory BIST with FPGA equivalents. See Converting ASIC Memories , on page 38 for details. |
|--------------|--|

| | |
|-----------|---|
| Exclusion | Exclude ASIC test interfaces like test TAP, test or JTAG muxing, and BSD. Exclude digital interfaces for analog blocks like ADC, DAC, and voltage regulators. |
|-----------|---|

- Use the report syntax_check command to check the RTL syntax, and then use the report command to see the results. Checking for syntax errors up front reduces costly debugging later in the cycle.
- If you want a quick prototype to quickly run through and validate your design and constraints, your design can have black boxes.

However, if you intend to go all the way through board bring up, your RTL cannot contain black boxes, because you will not be able to generate a bit file with black boxes.

For netlist editing and other details on converting ASIC RTL, refer to [Converting ASIC Designs to FPGA, on page 33](#).

- IP

With the HAPS prototyping solution, you can import Synopsys DesignWare IP directly. This means that you can use the same IP as in the ASIC, and you will not need to verify it again, which reduces the time to bring up. See [Dealing with IP, on page 53](#) for details.

- Constraints and attributes

- You can automatically read many existing constraint formats when you implement the FPGA, like sdc, fdc, cdc, and idc. This allows you to reuse existing files without having to recreate the constraints. In particular, you can reuse ASIC sdc timing constraints, and read in upf files that specify power intent.
- The constraints spreadsheet interface provides a quick and convenient way to enter constraints and attributes.
- Save more time by using the Tcl find, expand, and collection commands to quickly find a set of objects, group them, and apply constraints to them. You can also use the SDC query commands: get* and all*.
- Use the report constraint_check command to check the constraints, and then use the report command to see the results. Checking the constraints at this point is most efficient, because it reduces the amount of debugging later in the cycle, when it is costlier to retrace.
- For a quick prototype, set loose constraints.
- Avoid setting attributes like syn_keep that could inhibit optimizations.

See [Adding Design Files, on page 96](#) for more information.

Compiling a Quick Prototype

The following procedure outlines the steps needed to generate a quick prototype using command-line commands. You can also put these commands into a script to save you even more time.

As an alternative to the command line, you can also use the GUI to specify your design intent and create a quick prototype. See [Using Design Intent to Synthesize a Quick Prototype, on page 493](#).

1. Set up your RTL, taking advantage of the features that shorten design preparation that are mentioned in [Setting up the Design, on page 967](#).
2. Run the diagnostic compiler (`report rtl_diagnostics`) to make sure your RTL is clean.

This command parses the RTL and checks for syntax errors. By running this syntax check up front, you save time and debugging iterations later on in the flow.

- Check the diagnostic report and fix any errors.
- Rerun the diagnostic compiler with the `-incr 1` option and repeat the process until your RTL is clean. With this option, the diagnostic compiler runs incrementally, which saves time on subsequent runs.

3. Set options for, and run the Time-to-first-prototype (TTFP) fast compiler.

The following options are some of the options that you can use to reduce runtime. You can also set these options from the GUI by enabling them in the Options Editor ().

- In a Tcl options file, set option `set continue_on_error 1` to ensure that the compiler continues to run when it encounters a non-syntax error. The `continue_on_error` option saves runtime by reducing the number of iterations needed. If modules with errors need more work, black box them and work on them independently while taking the rest of the design through the flow. See [Running Continue on Error During Compilation, on page 821](#) for details.
- If you are using gated clock conversion, add the option `set fix_gated_and_generated_clocks 1` to the options file, to automatically convert ASIC gated and converted clocks.
- Specify the option `set automatic_compile_point 1` command to automatically define compile points. If you do not want a module to be made into an automatic compile point, set the `syn_no_compile_point` attribute on that module. If you have multiple licenses and specify the `max_parallel_jobs` option, the tool can reduce runtime further by processing the compile points in parallel on multi-core machines.
- Specify the TTFP compiler by setting the option `set synthesis_strategy fast|routability` command. The TTFP compiler is designed for fast

turnaround. The compiler generates a schematic which you can view to graphically analyze the design.

- Source the options file.
- Specify the run compile command.

The tool runs the TTFP compiler and generates a schematic and creates a new compile database state.

4. Analyze the results and fix errors.

- Use the view schematic command to view the results graphically and visualize the RTL, IP design hierarchy, and interconnect.
- Use the report command to check the results in the log file.
- For modules that were identified by the Continue on Error feature (CoE) and not compiled because of errors, work on them in isolation and fix the errors so that they can compile.
- Click error messages to get information on fixing them. Check resource utilization, clock timing, I/O timing, and critical paths to make sure that the constraints are feasible. Adjust constraints if needed.
- Fix any compilation errors.
- Iterate and run the TTFP compiler again. The point of iterating is to clean up the RTL, and the TTFP compile facilitates this by providing quick spins. You can save even more time by rerunning the TTFP compiler incrementally with the run compile -incr 1 command.

You can leverage the flexible database model and create different compile database states with each new run, or choose to overwrite a previously created database state. The database model lets you stay flexible and iterate through tight design loops for a particular design state.

- Repeat until there are no errors.

Synthesizing, Placing, and Routing a Quick Prototype

Once you have clean RTL and a compiled design, you can complete the synthesis process and run place-and-route. For an initial prototype, you might not want to go beyond this stage because your purpose is exploration or basic validation. However, you can continue on to hardware verification if your design is complete. The following steps provide the details:

1. Start with a clean compiled database state and run pre-map.
 - Set any options you might need in an options file. It is recommended that you use option set continue_on_error 1.
 - Go to the compile database state with the clean RTL (database set) and run the constraint checker (run constraint_check) to check pin I/Os and locations.
 - Use the run pre_map command. This command generates a pre-map database state, a schematic, and a log file.
 - Use the schematic and log file (view schematic and report) to analyze the constraints in addition to the synthesis results, and adjust the constraints if needed. You can save debugging time by using the SCOPE interface to adjust constraints, because it is linked to the schematic and the RTL.
 - Rerun run pre_map, leveraging the database model as before to create multiple database states if you are exploring different design options. You can also use run pre_map incr 1 to rerun it incrementally to retain design stability while also getting a faster turnaround time.
2. Map the design.
 - When you are satisfied with the pre-map results, select the pre-mapped database (database set) and map the design with run map. This command generates a mapped database state, a schematic, and a log file.
 - Analyze the schematic and log (view schematic and report), and fix any errors. Rerun run map, using the database model as before to explore the effects of different options. If you use run pre_map incr 1, the command runs incrementally and saves more time.
3. Run integrated place and route.
 - Select a mapped database state and use the run partition command to run place and route. This command runs place and route and generates a routed database state.
 - If you encounter errors, analyze and fix the errors before rerunning place and route incrementally.

If your quick prototype was intended to validate RTL feasibility or for design exploration, you can stop at this point. If you want to generate a bit file and validate the design on a board system, remember that your design must be complete with I/O definitions, and you cannot have black boxes.

4. To use the hardware system to verify the prototype, follow these steps:
 - Make sure your routed design is complete, with no black boxes.
 - Use the `export runtime` command on a placed and routed database state to generate a bit file. For an initial prototype you can use a lower frequency.
 - Set up the hardware system. For details, see the hardware documentation.
 - Configure the hardware using the configuration commands described in the *System Configuration Software Handbook*.
 - Verify the design on the hardware system.

Tips for Reducing Runtime

- Set accurate design constraints, Use report constraint_check to check them.
 - Set realistic clock frequencies; do not overconstrain them.
 - Specify synchronous/asynchronous clock groups and false paths.
- Use distributed compile and distributed synthesis.
- Black-box parts of the design to isolate them and reduce overhead.
- Use incremental flows where possible. Use the ECO flow for small changes. For place and route, use design checkpoint files (DCP) for incremental runs.
- Use multi-FPGA DTD3 and move debug logic from design FPGAs to the debug hub.
- Partition the design across as many FPGAs as possible, while balancing performance and reduced area utilization numbers.
- Disable area estimation when you run pre-partition.
- Balance area utilization across FPGAs using the exploratory approach with the TSS and PCF.
- Work with inter-FPGA connections, trace usage reports, and cable connections to reduce global route ratios so that you do not have everything routed.

Generating a Best-Performance Prototype

Typically, if the focus for your prototype is performance or optimal QoR, you have created initial prototypes previously. The following procedures are based on the assumption that you have previously worked on initial prototypes and have clean RTL and a feasible design.

Some scenarios where you want a performance-focused prototype are for an IP implementation, benchmarking, or to achieve timing closure.

- [Setting up for a Best-Performance Prototype](#), on page 973
- [Procedure for Creating a Best-Performance Prototype](#), on page 973

Setting up for a Best-Performance Prototype

The best-performance methodology described here assumes that you have cleaned the RTL and checked constraints as part of working on an initial prototype. The best-performance methodology starts with a compiled database state, and does not focus on the initial stages of migrating the design and compiling the design.

The following are some setup guidelines to achieve the best performance in the prototype:

- Set accurate and complete constraints.
- Reduce congestion by not using control signals with small fanouts.
- Avoid or reduce high fanout.

Procedure for Creating a Best-Performance Prototype

The following procedure outlines the steps needed to generate a performance-based prototype using command-line commands.

As an alternative to the command line, you can use the GUI to quickly specify your design intent to create a prototype with better performance. See [Synthesizing for Optimal QoR Using Design Intent](#), on page 494. However, the GUI method only covers synthesis, and does not include instrumentation and debug, as described here.

1. Generate an initial prototype to clean up the RTL and check design feasibility.

See [Methodology for a Quick Prototype, on page 966](#) for details, paying special attention to the information on setup and compiling. The following are some guidelines for setting up the RTL and constraints:

- Set accurate and complete constraints. Set feasible constraints.
- Reduce congestion by not using control signals with small fanouts.
- Avoid or reduce high fanout.

Based on your initial prototype, you might want to add test instrumentation to debug your RTL. Start with step 2 if you want to instrument a pre-compiled netlist. If you want to compile your design before adding instrumentation, go to step 3.

2. To add instrumentation to a pre-compiled netlist, start with the root database and create a new database state for instrumentation.
 - Use the `run pre_instrument` command, and specify the design files. The tool creates a new database state for RTL debugging.
 - Use the `edit idc -mode gui` command to open the GUI and define watchpoints and breakpoints. You can also do this through a Tcl script with the `edit idc -mode tclbatch` command.
 - Save the idc instrumentation file. You can now compile the design.

For more information about instrumenting the design, see [Instrumenting the Design for Debug, on page 641](#). For complete details, refer to the help built into the instrumentor.

3. Create a new compiled database state from the root database. If you instrumented a pre-compiled netlist in step 2, start with that database state.
 - Set options like the following for better performance:

```
option set enable_prepacking 1
option set fix_gated_and_generated_clocks 1
option set retiming 1
```
 - Run the `run compile` command. If you instrumented your design in step 2, include the `-idc` option and specify the idc instrumentation file. This command runs the default compiler, and generates a new, compiled database state. It also generates a log file and a schematic.
 - Analyze the results (view schematic and report list) and fix any errors. If you instrumented your design, the results will reflect this.

- Iterate as needed, using the `-incr 1` option for incremental compile runs. As you are starting with the results of the initial prototype, there should not be many iterations.
 - If you want to instrument the design after compilation, use the `-out` option and name a database state. For example: run `compile -out pclInstrument`.
4. To instrument a post-compiled netlist, follow these steps:
 - Generate a compiled database state for instrumentation as described in the previous step.
 - Use `edit idc -mode` to open the instrumentor, and instrument the design as described in step 2.
 - Save the `.idc` file.
 5. Synthesize, place, and route the design.
 - Set any options you require and run `re-map` (`run pre_map`). The command generates a pre-mapped database state, as well as log file and a schematic.
 - Analyze the results. Check the log file for the clock report, especially the gated and generated clocks. The prototyping tool is tightly integrated with IP, especially DesignWare IP, so that it can use accurate timing models for the IP and even optimize inside the IP in certain cases.

Iterate as needed. Isolate areas with problems and work on them independently to save time while you run through the rest of the design.

 - Map the design with `run map`. The command runs timing-driven synthesis with various optimizations. The tool intelligently maps the logic to available resources, including DSPs, RAMs, and LUTs. The results are closely correlated to final timing, because they are optimized for the target architecture and routing.
 - The `run map` command also generates a log file and a schematic that you can use to analyze the results and make adjustments. Iterate as needed. You can view the results of timing adjustments by running the `report timing` command and then checking the results in the log file.
 - Place and route the mapped design by running the `run partition` command. This command launches and runs place and route from the prototyping tool.

- Analyze the results and iterate through placement and routing until the design meets your goals. If necessary, you can return and rerun a synthesis step based on backannotation from place and route before working through the flow again.
- Generate a bit file by specifying the `export runtime bitfile` command. The command creates a bit file, and it also creates a directory with all the files you need for debugging your design, including a `debug.prj` file.

At this point, you can run RTL debug using the instrumentation you inserted earlier (step 6), or proceed to hardware verification (step 7).

6. If you want to debug the RTL, follow these steps:

- Start the debugger by typing `protocompiler_runtime debug -in path/debug.prj` at a system prompt. The `debug.prj` file is created when you generate the bit file.
- In the debugger window that opens, analyze and debug the RTL, based on the instrumentation you inserted earlier.
- Fix any errors by back-tracking to the appropriate database state, and rerunning the design flow from that point on. The flexible database model makes this easy to do. Generate a bit file when you have finished.

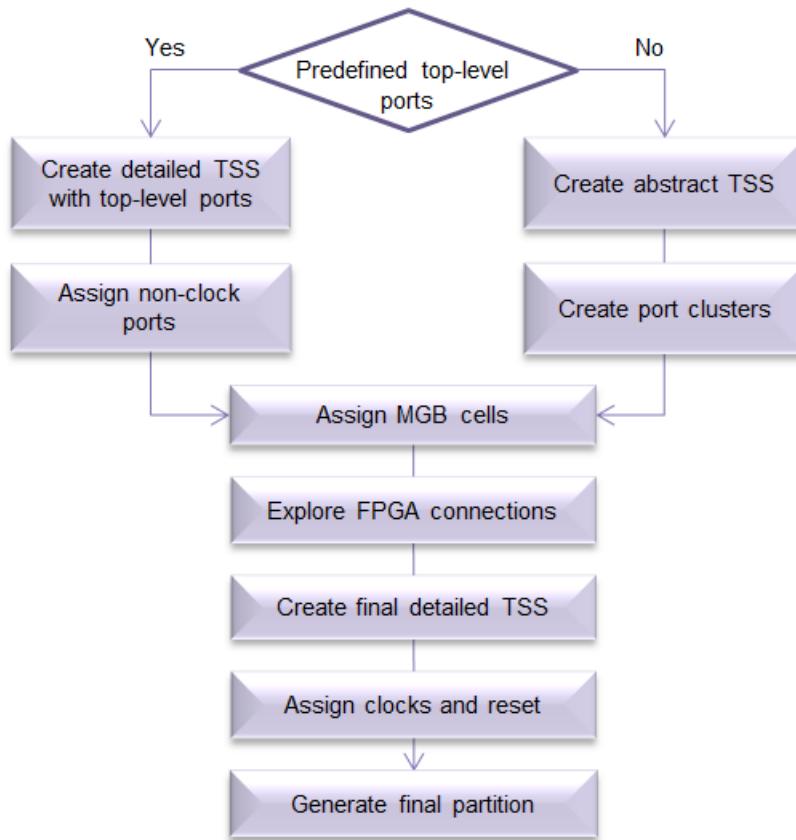
For more information about running RTL debug, see [Running the RTL Debugger, on page 681](#). For complete details, refer to the help built into the RTL debugger.

7. To verify the prototype using the hardware, follow these steps:

- Make sure your routed design is complete, with no black boxes.
- Use the `export runtime` command on a placed and routed database state to generate a bit file. For an initial prototype you can use a lower frequency.
- Set up the hardware. For details, see the hardware documentation.
- Configure the hardware system using the configuration commands described in the *System Configuration Software Handbook*.
- Verify the design on the hardware system.

Exploring Different Partitioning Solutions

The following figure summarizes the process of exploring and refining partitioning solutions as you iterate through this phase of the design and assign logic to FPGAs.



1. The first step is to assign top-level SoC ports. Some prototype implementations might have few external interfaces other than the FPGA I/Os.
2. The abstract target system specification (TSS) models the hardware configuration when no top-level ports are defined, specifying daughter boards and interconnect.

3. Use pcf file constraints to define port clusters and other information like bin utilization, hierarchy control, and floorplanning. You can also assign ports to the multi-gigabit (MGB) HAPS connector sites, because these high-speed I/Os are scarce resources.
4. Iterate between pre-partitioning and partitioning until you get the minimum performance you want and the prototype implementation is satisfactory.
5. Convert the pcf constraints into detailed TSS file commands. Assign system clocks and resets in the pcf file.

Run system route using the detailed TSS file and the system clock definitions.

Scripting Design Tasks

The software comes with some basic scripts for running synthesis. The contents of the scripts are described in the Reference Manual. You can use these scripts as templates and modify them to create scripts of your own to run design tasks.

Leveraging Examples and Templates

The `install_dir/examples` directory contains some scripted examples and templates that you can run or adapt for your own designs:

- `hpw_examples` contains examples, including the standard synthesis flow
- `hpw_templates` contains a template that you can fill in to create your own script

The template includes commands like the following:

```
set OPTION_FILE optionFile
set SRC_FILE_LIST sourceFile
set CONSTRAINT_FILE constraintFile
database create dbName
source $OPTION_FILE
run compile -filelist $SRC_FILE_LIST
run pre_map -filelist $CONSTRAINT_FILE
run map
run partition
export runtime -path outputDir
```

Leveraging Provided Scripts

See the following scripts described in the *Reference Manual*:

- FPGA Synthesis Scripts
- Scripts for Exploring Multiple Databases
- Debug Scripts

In addition to scripts, the FPGA prototyping tool supports Tcl commands that help you traverse the database and identify objects. You can use these commands to run batch conversions of ASIC RTL or for custom database access and reporting.

| Tcl Command | Description |
|-------------|---|
| find | Finds objects in the RTL database. Offers powerful sifting capabilities when combined with -filter. |
| expand | Finds objects by traversing the design hierarchy |
| collection | Creates groups of objects based on common attributes, that can then be manipulated together |

See [Chapter 1, Specifying Constraints](#) in the *Compiler Mapper Guide* for details about these commands.

Index

Symbols

DefaultLibraryMapping 111
_ISOLATE_LIB_ 117
_SYN_COMPATIBLE_INCLUDEPATH_ 116
USELIBGROUPORDER 116
.synopsys_pc.setup 173

A

MMC. 765
ACPM mode 373
adapter card
 specifying in TSS 207, 209
allow_duplicate_modules 94
APTN. *See* partitioner, partitioning
archive file
 extracting 66
archives
 creating 67
 extracting files from 71
 lite 74
 unarchiving 71
area estimates
 annotating to top level 435
 updating top level from lower level
 results 435
area estimation
 initial 325
ASICs 33
 clocking different from FPGA 42
 converting for FPGA prototyping 34
ASICs, converting
 power considerations 827
 UPF file 829
assignments.pcf 353
assignments.pcf, keeping clock
 assignments 351

asynchronous sets/resets
 and gated clock conversion 875
attributes in .cdc file 168
auto_export_reports option 515
auto_infer_blackbox 93
auto_replication.pcf 349
auto-cabling 197
automatic_compile_point 93, 479

B

backannotation
 data in system-level timing report 547
 top level with data from FPGAs 435
bind statement 911
bindandforce option 912
bins
 defined 188
 definition 241
bit file 614
black 526
black boxes
 continue on error 823
 distributed compile 505
 gated clock attributes 881
 in synthesized netlist 526
blocking-style license queuing 605
board bring-up 621
board bring-up utilities
 tssgen 226
board connectors
 mapping device pins 475
board system
 parameters 192
board_system_create commands
 syntax help 185
bookmarks
 using in log files 551

bottom-up compile 90
BRAM
 debug memory 743
 IICE buffer type 658
bring-up 621
buffer types
 IICE 657
built-in DTD 711

C

cable delay. *See* trace delay
cables
 defining 197
CAPIM_UI
 MDM 443, 445, 460
CAPIMs 452, 467
 checking 440, 452, 467
 sharing in MDM 452, 467
CCMs 764
cdc file
 specifying attributes and directives 168
cdc file syntax 169
CDPL 500
 for place and route exploration 577
cfg* commands 623
check_hstdm_timing
 syntax 532
check_timing_hstdm
 using 531
clock buffers
 replicating 349
clock constraints
 accounting for when partitioning 342
clock control module. *See* CCMs.
clock gate replication 349
clock managers. *See* CCMs, MMCM
clock replication 349
clock skew
 reducing through GCLK network 275
 replicating clock trees 349
clock sources 195
 FPGA 272
 PLL 271

clock synchronization, HAPS-70 and HAPS-DX7 290
clock_gate_replication_control command (pcf) 349
clocks
 converting ASIC to FPGA 42
 defining 44, 46
 defining for IICE 659
 HAPS 450, 465
 pre-assigned 351
 synchronizing HAPS-70 and HAPS-DX7 290

closed database. *See* obfuscated files
coding guidelines
 Formality 939
CoE. *See* continue on error 820
compilation 77
 definition 77
compilation process 822
compile
 incremental. *See* incremental compile
compiler
 compared to unified compiler 78
 default compiler 80
 diagnostic compiler 85
 fast compile mode 86
 methodology for using different modes 87
 modes 79
 multiple runs 87
 native. *See* compiler, default
compiler errors
 continue on error 821
compiler options 92
compilers
 TTFP. *See* TTFP compiler
compiling
 definition 77
Confpro
 clock definition 46
 environment variable 31
 project file 620
confpro 620, 621
congestion 589
connections
 cabling 197

-
- connector distribution, reducing
 congestion 594
 constraint checker 170
 constraints
 applied to multiple FPGAs 367
 constraints editor 474
 continue on error 820
 analyzing errors 822
 compilation 821
 flow 820
 reporting 822
 continue_on_error
 compiler option 94
 convert2protocompiler script 66
 corruption
 in sequential elements 846
 create_power_domain
 examples 831
 cross-module referencing 913
 custom scripts
 creating for synthesis 430
 FPGA synthesis 432
 Vivado place-and-route 432
 custom Vivado script 428, 429
 cut clocks 275
- D**
- database
 closed. *See* obfuscated files
 creating 58
 creating from Synplify Premier
 projects 65
 definition 58
 errors 62
 loading 62
 database hierarchy 59
 database node. *See* database state
 database query_state command 520
 database state 59
 green 62
 red 62
 database states
 comparing 522
 rerunning with out-of-date check 522
 databases
 accessing reports 512
 archiving 66
 compiling bottom-up 90
 naming 61
 overwriting 64
 querying 520
 querying option settings 521
 removing 64
 restoring from an archive 71
 stitching from the bottom-up 90
 datapath
 Datapath Conversion for Latches 905
 latch enable 905
 datapaths
 converting 905
 daughter boards
 See daughter cards
 adding 205
 custom 215
 defining 204
 finding supported 205
 guts.v file 218
 listing 191
 specifying in TSS 207
 daughter cards
 custom 191
 defining interconnect 201
 trace names 208
 TSS examples 208
 daughter cards. *See* daughter boards
 daughter cards. *See* daughter boards
 daughter cards. *See* daughter boards
 dc_root 94
 dcp file, generating congestion map 592
 DDR3
 built-in memory (HAPS-80) 658
 HAPS-DX7 hub for debug 659
 TSS connection example 210
 DDR3 daughter boards
 specifying in TSS 206
 DDR4
 memory card for debug 658
 debug
 instance insertion 911
 debug flow
 HAPS-70 716
 HAPS-70 single FPGA 707

debug flows
 HAPS-80 single FPGA 703

debugger 681

debugging
 performance 743
 using BRAM 743

deep trace debug
 multi-FPGA 718

define_haps_io 475, 487
 differential pairs 483
 HAPS-80 287
 using variables 482

delays
 HAPS-80 HSTDIM 388
 input/output 385
 trace 394

design
 compiling 77

design checkpoint file (dcp) 582

Design Compiler
 avoiding DesignWare IP black boxes 526

Design Compiler environment variable 31

design flows
 flow diagrams 22
 implementation mode 29
 multi-FPGA flow diagrams 20
 partition mode 29
 partitioning 178
 single-FPGA flow diagram 19

design intent 493
 fast turnaround 493
 QoR 494

design_timeest.est 346

DesignWare
 avoiding black boxes for IP 526
 license queuing 607

diagnostic compiler
 using 85, 87

differential clocks 486

differential pairs 483
 differential paris 483

directives in .cdc file 168

distributed compilation. *See* distributed compile

distributed compile 503
 incremental 505

distributed processing 497
 CDPL 500
 compilation 503
 operations 497
 options 498
 schemes 497
 scripts for synthesizing FPGAs 427
 synthesis 506

distributed synthesis 506
 incremental run 508

drivers
 replacing ASIC drivers 911

DTD
 built-in on HAPS systems 711

DTD3 740

DTD4 711

dual-signal retention 849, 854

dw.foundation 94

dw_minpower 94

dw_stop_on_nolic 94

E

ECO flow
 place and route 583

EDIF
 including for unified compile 131

enable_prepacking 94

encrypting source files 655

environment variables 30

equivalence checker. *See* Formality

error codes 604

error messages 550
 gated clock report 895
 in files that cannot be exported 550

errors
 black boxing 822
 continuing 820
 downgradable 552, 555
 generating report 555

ESDB

generating 797
 instrumenting 798
 essential signal database. *See* ESDB
estimate_timing
 and optimization_priority 375
export runtime 615
export vivado command 581
 external clock distribution board. *See* ECDB

F

fanout
 effect on congestion 590
fast clock simulation 572
fast compiler. *See* TTFP compiler
fast sample clock. *See* oversampling
fast signal database. *See* FSDB
fast turnaround
 design intent 493
 distributed compilation 503
 distributed synthesis 506
 prototypes 966
fast_proto_mode 94
FDC
 clock definition overview 45, 47
files
 encrypting source 655
 idc 648
 Tcl 609
See also Tcl commands
 Tcl batch script 603
five_fixed_traces 248
force statement 911
force_async_genclk_conv example 887
formal verification. *See* Formality
Formality
 guidelines 938
 launch command 948
 running 937
 unverifiable modules 960
 using for multi-FPGA verification 953
 using for single-FPGA verification 945

verification mode option 946
FORMALITY variable 946
FPGA implementation flow
 design flow 472
 details 57, 471, 819
FPGAs
 generating individual 416
 implementing with a script 426
FSDB 575, 799

G

gated clock conversion 862
 combinational logic examples 879
 constraint examples 881
 definition 863
 forcing conversion with asynchronous set/reset 875
gated clocks
 attributes for black boxes 881
 conversion requirements 873
 converting 862
 definition 862
 effect on congestion 590
 error messages 895
 procedure for fixing 870
GCC. *See* gated clock conversion
GCC. *See* gated clock conversion, gated clocks, generated clocks
GCLK. *See* global clocks
GCLK0 486
 working with 269
gclk0 location constraints 486
GCLK8 trace group 248
generated clock conversion 862
 examples 885
generated clocks
 converting 862
 definition 862
 guidelines 865
global clocks 269
 assigning FPGA clocks 272
 assigning PLLs 271
 defining 475
 source 270
global default

viewing setting 174
global reset synchronization 722
global state visibility. *See* GSV
GPIO
 HAPS-80 286
 TSS connection example 212
GPIOLINK 488
graph model for partitioning 343
group mapping 109
 file description 110
 file example 113
GSV
 advantages 753, 803
 asynchronous 755
 configuring clock control modules 764
 synchronous 755
 using asynchronous GSV 757

H

HAPS awareness 475
HAPS connectors
 mapping 475
HAPS I/O report 477
HAPS pin mapping 475
HAPs pins 482
HAPS software and hardware system 16
haps_io_report 482
HAPS-100
 HSTDMD training 377
 initializing 377
 multi-design mode (MDM) 437
 using with HAPS-80 601
HAPS-70
 debug flow 716
 hstdm 382
 incorporating HAPS-DX7 289
 MDM 437
HAPS-70 debug flow
 single FPGA 707
HAPS-80
 architecture overview 284
 GPIO 283
 HSTDMD 380
 HSTDMD delay table 387
 MDM 437
memory 285
 using with HAPS-100 601
HAPS-80 debug flow
 single FPGA 703
HAPS-DX7
 using chained board for debugging 718
 using with HAPS-70 289
hapsinit.sim module 561
hapsinit.vm file 562
hardware bring-up 621
hardware configuration
 defining 184
 verifying for debug 690
HDL bypass flow 165
HDL source
 including 655
hdl_define 92
 in lmf file 111
hierarchical partitioning 406
high speed time domain multiplexing.
 See HSTDMD
hop 344
HSTDMD 373
 checking results 531
 circuit training 383
 delay 384
 force statements 569
 HAPS-controlled training 380
 reset hijack training 382
 running VCS simulation 568
 simulation factors 571
 single-ended 380, 387
 HSTDMD delay
 single-ended 388
 HSTDMD delays
 HAPS-80 388
 HSTDMD ERD mode 373
 HSTDMD training
 HAPS-100 377
 simulation 572
 HTML reports 514
 hyper source
 example 917
 for IPs 915
 for prototyping 915

-
- threading signals 916
I
 I/O planning 482
 I/O report
 HAPS pin mapping 477
 ICG 867
 stability latch removal 875
 idc file
 editing 648
 IEEE 1801-2009. *See* UPF
 IICE
 adding 652
 asynchronous clocks 660
 buffer type 657
 clocks 659
 common parameter settings 654
 defined 651
 deleting 653
 individual parameter settings 655
 multi-clock 660
 sample clock 659
 include_path 95
 incremental compile 79
 default compiler 85
 diagnostic compiler 85
 fast compile 86
 HDL bypass flow 165
 incremental flow
 Xilinx place and route 582
 incremental synthesis
 distributed 508
 instance insertion for debugging 911
 instrumentation
 post-compile 648
 instrumentor
 launching 642
 running 641
 running after compilation 648
 Integrated Clock Gating (ICG)
 stability latch removal 875
 integrated clock gating cells. *See* ICG
 interconnect
 daughter card 201
 inter-FPGA timing analysis 547
 IP 53
 integrating 53
 license queuing 606
 revalidating 53
 IP license queuing 606
 IPs
 using hyper source for debug 915
 is_error_blackbox property 824
 ISOLATE_LIB 117
 isolation power strategy 835
 It 265
 iterations
 reducing with continue on error 820
J
 jobs
 querying 523
L
 launch vivado command 581
 libext 95
 library mapping 109
 library_path 95
 license queuing 604
 blocking-style 605
 DesignWare IP 606
 IP 606
 license release (synthesis)
 after P&R 607
 license_release 607
 LM_LICENSE_FILE 27
 lmf file. *See* group mapping
 log file
 continue on error 822
 log files
 accessing 512
 unified compile (uc.log) 122
 looplimit 95
 LUT combining 94
M
 macros, Verilog 110

- map
 - using UPF 830
 - map_retention_cell command 854
 - mapped netlist
 - verifying against original RTL 945
 - verifying against partitioned RTL 953
 - mapping
 - simulating after 560
 - max_parallel_jobs 95, 498, 499
 - max_parallel_jobs option 612
 - max_parallel_jobs variable 612
 - maximum parallel jobs 612
 - MDM 437
 - CAPIM sharing 452, 467
 - HAPS-100 437
 - with multi-FPGA synchronous GSV 761
 - memory
 - for debug 703, 708
 - memory usage
 - schematics 518
 - message override 552
 - message overrides
 - in setup file 553
 - message_override command 552
 - messages
 - display 551
 - suppressing 551
 - metrics
 - querying 524
 - MGB cards
 - specifying in TSS 207
 - MGB2 401
 - MGTDM 401
 - mixed designs
 - black boxes 526
 - mixed designs, mixed systems 601
 - mixed hardware, HAPS-100 and HAPS-80 601
 - MMCM
 - GSV 765
 - MPF
 - Vivado 434
 - multi-clock IICE 660
 - multi-design mode. *See* MDM 437
 - multi-FPGA deep trace debug 718
 - multi-FPGA design flows 20
 - multi-FPGA designs
 - running Formality 953
 - multi-FPGA timing analysis 547
 - multi-hop path 344
 - multi-hop paths
 - reports, after run system_route 346
 - reports, post-partition 345
 - multiprocessing 95
 - maximum parallel jobs 612
 - multiprocessing. *See* distributed processing
 - multi-use mode. *See* multi-design mode
 - multi-user mode. *See* multi-design mode
- ## N
- name matching
 - post-mapped timing reproto 533
 - netlist editing 41
 - procedure 907
 - sample file 908
 - nets
 - cdc directives 169
 - NGC
 - including for unified compile 131
- ## O
- obfuscated files
 - accessing 515
 - database 59
 - offline viewing 514
 - optimization_priority
 - and estimate_timing 375
 - option 107
 - options 171
 - customizing based on design intent 494
 - customizing for fast turnaround 493
 - default setting 107, 174
 - fast turnaround 493
 - QoR 494
 - querying settings 521

-
- script for current values [521](#)
 - setting on startup [173](#)
 - viewing current value [174](#)
 - viewing default value [174](#)
 - options file [171](#)
 - original source
 - including for debug [655](#)
 - oversampling
 - IICE [660](#)
 - P**
 - package library, adding [102](#)
 - parallel code [911](#)
 - parallel jobs [612](#)
 - distributed compile [504](#)
 - parallel processing. *See* distributed processing
 - parameterized modules
 - instrumenting [649](#)
 - partition reports
 - abstract partition iterations [363](#)
 - final partition [364](#)
 - initial partition [361](#)
 - second partition [362](#)
 - system route [365](#)
 - partitioner [180, 421](#)
 - partitioning [179](#)
 - analyzing reports [181](#)
 - based on timing graph [343](#)
 - checking reports [367](#)
 - design flow [178](#)
 - exploring choices [242](#)
 - files [183](#)
 - generating individual FPGAs [416](#)
 - hierarchical [406](#)
 - methodology [360](#)
 - parsing reports [370](#)
 - restrictions [333](#)
 - summary of controls [332](#)
 - timing aware [342](#)
 - timing report [548](#)
 - to single FPGA [335](#)
 - using run partition [328](#)
 - using run pre_partition [324](#)
 - with clock constraints [342](#)
 - partitioning reports
 - using [360](#)
 - partitioning stages [179](#)
 - partitions
 - analyzing [358](#)
 - freezing [352](#)
 - passwords
 - encryption/decryption [655](#)
 - PATH environment variable [27](#)
 - PCF
 - for clock definition [45](#)
 - pcf
 - assigning reset [383](#)
 - pcf constraints
 - specifying [241](#)
 - pcf file
 - clock gate replication [349](#)
 - two purposes [241](#)
 - pcf files
 - types [246](#)
 - PCIe [25](#)
 - pd_strategy property, UPF [859](#)
 - performance
 - using design intent [494](#)
 - phase-locked loops [865](#)
 - phase-locked loops. *See* PLLs
 - pin mapping
 - define_haps_io [475](#)
 - HAPS I/O report [477](#)
 - pin multiplexing [372](#)
 - place and route [576](#)
 - black boxes [526](#)
 - exploration [576](#)
 - exporting database [581](#)
 - reducing congestion [589](#)
 - Verilog netlist [581](#)
 - PLLs
 - defining as clock source [271](#)
 - HAPS-80 [271](#)
 - port mismatches
 - black boxes in synthesized netlist [526](#)
 - port types
 - supported in SLP [333](#)
 - post-compile instrumentation [648](#)
 - post-partition netlist
 - verifying [953](#)

- power domains, UPF [830](#)
 - power_domain property, UPF [859](#)
 - pragmas
 - for unified compile [127](#)
 - pre-assigned clocks [351](#)
 - pre-map
 - using UPF [830](#)
 - pre-partitioning [324](#)
 - prf file [367](#)
 - project files
 - VHDL library [102](#)
 - proto_rt
 - using [377](#)
 - protocompiler command [28](#)
 - ProtoCompiler DX [16](#)
 - protocompiler_dx command [28](#)
 - prototypes
 - fast turnaround [966](#)
 - optimizing timing [973](#)
 - QoR [973](#)
 - prototyping [33](#)
 - converting ASIC designs [34](#)
 - guidelines [35](#)
 - introduction [16](#)
 - using hyper source threading [915](#)
 - prototyping modes [24](#)
 - prototyping with PCIE [25](#)
- ## **Q**
- QoR
 - prototypes [973](#)
 - QSFP boards
 - specifying in TSS [207](#)
 - QSFP cards
 - TSS connection example [213](#)
 - queries (database query_state) [520](#)
 - query metrics [524](#)
- ## **R**
- RAM
 - for deep trace debug [708](#)
 - UPF retention strategy [852](#)
 - RAMs
- formal verification considerations [941](#)
 - registers
 - UPF retention strategy [850](#)
 - report constraint_check [170](#)
 - report files
 - text [515](#)
 - report message command [555](#)
 - reports
 - accessing [512](#)
 - exporting [514](#)
 - messages [555](#)
 - resource usage [55](#)
 - viewing [513](#)
 - viewing offline [514](#)
 - reset
 - assigning [383](#)
 - resets
 - MDM [449, 464](#)
 - synchronizing [280](#)
 - resource utilization [55](#)
 - retention
 - RAMs [852](#)
 - schemes [849](#)
 - retention directives [854](#)
 - retention polarity [850](#)
 - retention power strategy [847](#)
 - return codes [604](#)
 - riser card
 - specifying in TSS for daughter boards [207](#)
 - router_constraints.pcf [353](#)
 - routing
 - system level [412](#)
 - RTL bypass flow [165](#)
 - RTL bypass flow. *See* HDL bypass flow
 - run par_explorer command [577](#)
 - run system_route
 - effect of options on TDM [375](#)
 - running P&R
 - license release (synthesis) [607](#)
 - runtime
 - continue on error [820](#)
 - incremental place and route [582](#)
 - reducing [972](#)

speeding up with distributed compilation 503
speeding up with distributed synthesis 506

S

sample clock
 IICE 659

sampling signals 654, 746

sar archive file 66

SATA cabling 722

schematic
 continue on error 823

schematics
 memory usage 518
 viewing results 517

scripts 979
 customizing. *See* custom scripts
 implementing individual FPGAs 427

SE. *See* single-ended HSTDM

search order 99
 __USELIBGROUPORDER__ 116

See also CCMs

sequential shifters
 UPF retention strategy 853

set_datapathonly_delay 419

set_retention_control command 849, 854

set_scope 833

setup files 173
 message override options 553

signals
 sampling selection 654, 746
 threading with hyper source. *See* hyper source

Siloti 797

simulation 557, 560
 gate-level 560
 gate-level (multi-FPGA) 573
 HSTDM designs 568
 post-partition 567
 RTL simulation for multi-FPGA designs 562
 running 558

simulation mode 572

single-ended HSTDM 380, 387, 388

single-FPGA design flows 19

single-FPGA designs
 running Formality 945

:

single-FPGA partitioning 335
singl-signal retention 849
slack
 optimization 343, 344
SLRs, reducing congestion 594
SLTA 419
SLTA. *See* system-level timing analysisr
SoC 33
source files 97
 encrypting 655
 querying 521
 text file syntax 102
 using variables 105
source-level partitioning
 restrictions 333
speed grade
 adding for partitioned designs 186
 for single-FPGA designs 476
stability latches
 ICG 875
standalone prototyping mode 24
starting the tool 27
startup
 setting options 173
state machines
 retention control 854
superbins 408, 409
syn_black_box 93
syn_implement 854
syn_no_compile_point 969
syn_no_prune 93
syn_preserve 93
syn_rename_module 93
syn_ret_lib_cell_type 854
syn_ret_type 854
syn_sharing 93
syn_unique_inst_module 93
syn_upf_ret_port_type 854
synopsys_pc.setup options file 109, 173
Synplify Premier projects, converting 65
synthesis mode 429
system configuration

- environment variable 31
- system generate
 - timing report 548
- system-level routing 412
- system-level timing analysis 547
- system-level timing analysis (SLTA). *See* SLTA
- SystemVerilog
 - coding guidelines for Formality 940
 - referencing Verilog elements 914

T

- target system
 - detailed 189
- target system specification file. *See* TSS
- Tcl commands
 - batch script 603
 - running 609
- Tcl files 609
 - creating 611
- Tcl scripts
 - See* Tcl files.
- TDM
 - effect of run system_route options 375
 - modes 373
 - simulation factors 572
 - slack-based partitioning 343
- TDM ratio
 - effect on multi-hop paths 344
- tdm_control -type command 373
- tdm_modules.v 571
- tdm_qualified_timing.rpt 346
- testbench
 - generating for VCS simulation 575
- thin compiler. *See* TTFP compiler
- time budgeting
 - checking results 531
- timing
 - prototypes 973
- timing analysis 529
 - inter-FPGA 547
 - intra-FPGA 529
 - single FPGA 529
 - system-level 547
- timing analyst 541

:

timing budgeting 419
timing exceptions
slack-based optimization 343
timing graph 342
Timing Report view 536, 537
timing reports
 custom 541
 customizing 530
 generating 529
 system level 547
 Timing View 537
top_module 95
trace assignment
 daughter boards 208
 finding trace names 207, 359
 manual 265
trace delay 394
trace groups 248
trace names 265
 finding 265
 report target_system -tss 207
traces
 for daughter cards 208
 routing 412
triggers
 pre-armming 670
tsd file 619
TSS
 connection examples 208
 definition 184
 for clock definition 45
 specifying daughter boards 207
 specifying DDR3 daughter boards 206
tss
 and tsd 619
tss_list_traces
 finding trace names 265
TTFP compiler
 initial prototype 494
 using 86, 969

U

UDP files 100
UMRBus
 chaining 201

- signal connection 488
- unified compile 120
 - compared to native compiler 77
 - EDiF/NGC 131
 - instrumenting design 127
- Unified Power Format. *See* UPF
- Unified Use Model
 - See* UUM
- UPF 827
 - checking implementation 857
 - corruption 846
 - description 54
 - design flow 828
 - for unified compile 130
 - limitations 827
 - properties for schematic viewing 859
 - specifying isolation cells 835, 843
 - specifying power domains 830
 - specifying retention registers 847
 - verification 860
- UPF report
 - using 857
- utf file
 - supported commands 122, 129
- utilities
 - board bring-up 621
- UUM 109
 - compiling 99

V

- variables
 - fdc file for define_haps_io 482
 - for VCS simulation 558
 - in file paths 105
- VCS 557, 561
 - do script 563
 - generated files for simulation 563
 - simulating unverifiable components 960
- VCS_HOME variable 946
- Verdi
 - debugging flow 792
 - instrumenting ESDB signals 798
- Verdi nWave viewer 800
- verification
 - simulation. *See* simulation
 - UPF 860

:

verification mode option 946

Verilog

- coding guidelines for Formality 940
- netlist for Vivado 581
- referencing SystemVerilog elements 914
- standard 102

VHDL files

- adding library 102
- adding third-party package library 102

vhdl2008 95

Vivado 576

- customizing place-and-route scripts 432
- exporting database 581
- incremental flow after resynthesis 588
- MPF 434
- multi processing flow 434
- using custom script 428, 429
- Verilog netlist 581

vlog_std 95

VM_NETLIST_DIR

- VCS simulation 559

voltage regions 192

W

warnings

- generating report 555
- handling 550

waveform viewers

- Verdi 799

X

Xilinx

- incremental flow 582
- mapping pins to board connectors 475

XMR. See cross-module referencing