Intel® Quartus® Prime Standard Edition Handbook Volume 3

Verification

Updated for Intel® Quartus® Prime Design Suite: **17.1**

Subscribe

Send Feedback

QPS5V3 | 2017.11.06

Latest document on the web: PDF | HTML

Contents

1 Simulating Intel FPGA Designs

This document describes simulating designs that target Intel FPGA devices. Simulation verifies design behavior before device programming. The Intel® Quartus® Prime software supports RTL- and gate-level design simulation in supported EDA simulators. Simulation involves setting up your simulator working environment, compiling simulation model libraries, and running your simulation.

1.1 Simulator Support

The Intel Quartus Prime software supports specific EDA simulator versions for RTL and gate-level simulation.

Table 1. Supported Simulators

Vendor	Simulator	Version	Platform
Aldec	Active-HDL*	10.3	Windows
Aldec	Riviera-PRO*	2016.10	Windows, Linux
Cadence	Incisive Enterprise*	15.20	Linux
Mentor Graphics*	ModelSim* - Intel FPGA Edition	10.5c	Windows, Linux
Mentor Graphics	ModelSim PE	10.5c	Windows
Mentor Graphics	ModelSim SE	10.5c	Windows, Linux
Mentor Graphics	QuestaSim*	10.5c	Windows, Linux
Synopsys*	VCS* VCS MX	2016,06-SP-1	Linux

Table 2. Supported Simulators

Vendor	Simulator	Version	Platform
Aldec	Active-HDL	10.3	Windows
Aldec	Riviera-PRO	2015.10	Windows, Linux
Cadence	Incisive Enterprise*	14.20	Linux
Mentor Graphics	ModelSim - Intel FPGA Edition	10.5b	Windows, Linux
Mentor Graphics	ModelSim PE	10.4d	Windows
Mentor Graphics	ModelSim SE*	10.4d	Windows, Linux
Mentor Graphics	QuestaSim	10.4d	Windows, Linux
Synopsys	VCS VCS MX	2014,12-SP1	Linux

1.2 Simulation Levels

The Intel Quartus Prime software supports RTL and gate-level simulation of IP cores in supported EDA simulators.

Table 3. Supported Simulation Levels

Simulation Level	Description	Simulation Input
RTL	Cycle-accurate simulation using Verilog HDL, SystemVerilog, and VHDL design source code with simulation models provided by Intel and other IP providers.	• Design source/testbench • Intel simulation libraries • Intel FPGA IP plain text or IEEE encrypted RTL models • IP simulation models • Intel FPGA IP functional simulation models • Intel FPGA IP bus functional models • Platform Designer (Standard)-generated models • Verification IP
Gate-level functional	Simulation using a post-synthesis or post-fit functional netlist testing the post-synthesis functional netlist, or post-fit functional netlist.	• Testbench • Intel simulation libraries • Post-synthesis or post-fit functional netlist • Intel FPGA IP bus functional models
Gate-level timing	Simulation using a post-fit timing netlist, testing functional and timing performance. Supported only for the Arria® II GX/GZ,Cyclone® IV, MAX® II, MAX V, and Stratix® IV device families.	• Testbench • Intel simulation libraries • Post-fit timing netlist • Post-fit Standard Delay Output File (`.sdo`).

Note: Gate-level timing simulation of an entire design can be slow and should be avoided. Gate-level timing simulation is supported only for the Arria II GX/GZ,Cyclone IV, MAX II, MAX V, and Stratix IV device families.. Use Timing Analyzer static timing analysis rather than gate-level timing simulation.

1.3 HDL Support

The Intel Quartus Prime software provides the following HDL support for EDA simulators.

Table 4. HDL Support

Language	Description
VHDL	• For VHDL RTL simulation, compile design files directly in your simulator. You must also compile simulation models from the Intel FPGA simulation libraries and simulation models for the IP cores in your design. Use the Simulation Library Compiler to compile simulation models. • To use NativeLink automation, analyze and elaborate your design in the Intel Quartus Prime software, and then use the NativeLink simulator scripts to compile the design files in your simulator. • For gate-level simulation, the EDA Netlist Writer generates a synthesized design netlist VHDL Output File (`.vho`). Compile the `.vho` in your simulator. You may also need to compile models from the Intel FPGA simulation libraries. • IEEE 1364-2005 encrypted Verilog HDL simulation models are encrypted separately for each simulation vendor that the Quartus Prime software supports. To simulate the model in a VHDL design, you must have a simulator that is capable of VHDL/Verilog HDL co-simulation.
Verilog HDL -SystemVerilog	• For RTL simulation in Verilog HDL or SystemVerilog, compile your design files in your simulator. You must also compile simulation models from the Intel FPGA simulation libraries and simulation models for the IP cores in your design. Use the Simulation Library Compiler to compile simulation models. • For gate-level simulation, the EDA Netlist Writer generates a synthesized design netlist Verilog Output File (`.vo`). Compile the `.vo` in your simulator.
Mixed HDL	• If your design is a mix of VHDL, Verilog HDL, and SystemVerilog files, you must use a mixed language simulator. Choose the most convenient supported language for generation of Intel FPGA IP cores in your design. • Intel FPGA provides the entry-level ModelSim - Intel FPGA Edition software, along with precompiled Intel FPGA simulation libraries, to simplify simulation of Intel FPGA designs. Starting in version 15.0, the ModelSim - Intel FPGA Edition software supports native, mixed-language (VHDL/Verilog HDL/SystemVerilog) co-simulation of plain text HDL. If you have a VHDL-only simulator and need to simulate Verilog HDL modules and IP cores, you can either acquire a mixed-language simulator license from the simulator vendor, or use the ModelSim - Intel FPGA Edition software.
Schematic	You must convert schematics to HDL format before simulation. You can use the converted VHDL or Verilog HDL files for RTL simulation.

1.4 Simulation Flows

The Intel Quartus Prime software supports various simulation flows.

Table 5. Simulation Flows

Simulation Flow	Description
Scripted Simulation Flows	Scripted simulation supports custom control of all aspects of simulation, such as custom compilation commands, or multipass simulation flows. Use a version-independent top-level simulation script that "sources" Intel Quartus Prime-generated IP simulation setup scripts. The Intel Quartus Prime software generates a combined simulator setup script for all IP cores, for each supported simulator.
NativeLink Simulation Flow	NativeLink automates Intel Quartus Prime integration with your EDA simulator. Setup NativeLink to generate simulation scripts, compile simulation libraries, and automatically launch your simulator following design compilation. Specify your own compilation, elaboration, and simulation scripts for testbench and simulation model files. Do not use NativeLink if you require direct control over every aspect of simulation. *Note:* The Intel Quartus Prime Pro Edition software does not support NativeLink simulation.
Specialized Simulation Flows	Supports specialized simulation flows for specific design variations, including the following:

continued...

Simulation Flow	Description
	• For simulation of example designs, refer to the documentation for the example design or to the IP core user guide. • For simulation of Platform Designer designs, refer to *Creating a System with Platform Designer (Standard)* or *Creating a System with Platform Designer*. • For simulation of designs that include the Nios® II embedded processor, refer to *Simulating a Nios II Embedded Processor*.

Related Links

- IP User Guide Documentation
- AN 351: Simulating Nios II Embedded Processors Designs
- Creating a System With Platform Designer (Standard)
  In *Intel Quartus Prime Standard Edition Handbook Volume 3*

1.5 Preparing for Simulation

Preparing for RTL or gate-level simulation involves compiling the RTL or gate-level representation of your design and testbench. You must also compile IP simulation models, models from the Intel FPGA simulation libraries, and any other model libraries required for your design.

1.5.1 Compiling Simulation Models

The Intel Quartus Prime software includes simulation models for all Intel FPGA IP cores. These models include IP functional simulation models, and device family-specific models in the `<Intel Quartus Prime` *`installation path`*`>/eda/sim_lib` directory. These models include IEEE encrypted Verilog HDL models for both Verilog HDL and VHDL simulation.

Before running simulation, you must compile the appropriate simulation models from the Intel Quartus Prime simulation libraries using any of the following methods:

- Use the NativeLink feature to automatically compile your design, Intel FPGA IP, simulation model libraries, and testbench.
- To automatically compile all required simulation model libraries for your design in your supported simulator, click **Tools ➤ Launch Simulation Library Compiler**. Specify options for your simulation tool, language, target device family, and output location, and then click **OK**.
- Compile Intel Quartus Prime simulation models manually with your simulator.

Use the compiled simulation model libraries to simulate your design. Refer to your EDA simulator's documentation for information about running simulation.

Note: The specified timescale precision must be within 1ps when using Intel Quartus Prime simulation models.

Related Links

Intel Quartus Prime Simulation Models
In Intel Quartus Prime Pro Edition Help

1.6 Simulating Intel FPGA IP Cores

The Intel Quartus Prime software supports IP core RTL simulation in specific EDA simulators. IP generation creates simulation files, including the functional simulation model, any testbench (or example design), and vendor-specific simulator setup scripts for each IP core. Use the functional simulation model and any testbench or example design for simulation. IP generation output may also include scripts to compile and run any testbench. The scripts list all models or libraries you require to simulate your IP core.

The Intel Quartus Prime software provides integration with many simulators and supports multiple simulation flows, including your own scripted and custom simulation flows. Whichever flow you choose, IP core simulation involves the following steps:

1. Generate simulation model, testbench (or example design), and simulator setup script files.
2. Set up your simulator environment and any simulation scripts.
3. Compile simulation model libraries.
4. Run your simulator.

1.6.1 Generating IP Simulation Files

The Intel Quartus Prime software optionally generates the functional simulation model, any testbench (or example design), and vendor-specific simulator setup scripts when you generate an IP core. To control the generation of IP simulation files:

- To specify your supported simulator and options for IP simulation file generation, click **Assignment ➤ Settings ➤ EDA Tool Settings ➤ Simulation**.
- To parameterize a new IP variation, enable generation of simulation files, and generate the IP core synthesis and simulation files, click **Tools ➤ IP Catalog**.
- To edit parameters and regenerate synthesis or simulation files for an existing IP core variation, click **View ➤ Utility Windows ➤ Project Navigator ➤ IP Components**.

Table 6. Intel FPGA IP Simulation Files

File Type	Description	File Name
Simulator setup scripts	Vendor-specific scripts to compile, elaborate, and simulate Intel FPGA IP models and simulation model library files. Optionally, generate a simulator setup script for each vendor that combines the individual IP core scripts into one file. Source the combined script from your top-level simulation script to eliminate script maintenance.	`<`*`my_dir`*`>/aldec/rivierapro_setup.tcl` `<`*`my_dir`*`>/cadence/ncsim_setup.sh` `<`*`my_dir`*`>/mentor/msim_setup.tcl`

continued...

File Type	Description	File Name
		`<my_dir>/synopsys/vcs/vcs_setup.sh`
Simulation IP File (Intel Quartus Prime Standard Edition)	Contains IP core simulation library mapping information. To use NativeLink, add the `.qip` and `.sip` files generated for IP to your project.	`<design name>.sip`

IP functional simulation models (Intel Quartus Prime Standard Edition)	IP functional simulation models are cycle-accurate VHDL or Verilog HDL models a that the Intel Quartus Prime software generates for some Intel FPGA IP cores. IP functional simulation models support fast functional simulation of IP using industry-standard VHDL and Verilog HDL simulators.	`<my_ip>.vho`
		
		`<my_ip>.vo`

IEEE encrypted models (Intel Quartus Prime Standard Edition)	Intel provides Arria V, Cyclone V, Stratix V, and newer simulation model libraries and IP simulation models in Verilog HDL and IEEE-encrypted Verilog HDL. Your simulator's co-simulation capabilities support VHDL simulation of these models. IEEE encrypted Verilog HDL models are significantly faster than IP functional simulation models. The Intel Quartus Prime Pro Edition software does not support these models.	`<my_ip>.v`

Note: Intel FPGA IP cores support a variety of cycle-accurate simulation models, including simulation-specific IP functional simulation models and encrypted RTL models, and plain text RTL models. The models support fast functional simulation of your IP core instance using industry-standard VHDL or Verilog HDL simulators. For some IP cores, generation only produces the plain text RTL model, and you can simulate that model. Use the simulation models only for simulation and not for synthesis or any other purposes. Using these models for synthesis creates a nonfunctional design.

1.6.1.1 Generating IP Functional Simulation Models (Intel Quartus Prime Standard Edition)

Intel provides IP functional simulation models for some Intel FPGA IP supporting 40nm FPGA devices.

To generate IP functional simulation models:

1. Turn on the **Generate Simulation Model** option when parameterizing the IP core.
2. When you simulate your design, compile only the `.vo` or `.vho` for these IP cores in your simulator. Do not compile the corresponding HDL file. The encrypted HDL file supports synthesis by only the Intel Quartus Prime software.

   Note:
   - Intel FPGA IP cores that do not require IP functional simulation models for simulation, do not provide the **Generate Simulation Model** option in the IP core parameter editor.
   - Many recently released Intel FPGA IP cores support RTL simulation using IEEE Verilog HDL encryption. IEEE encrypted models are significantly faster than IP functional simulation models. Simulate the models in both Verilog HDL and VHDL designs.

Related Links

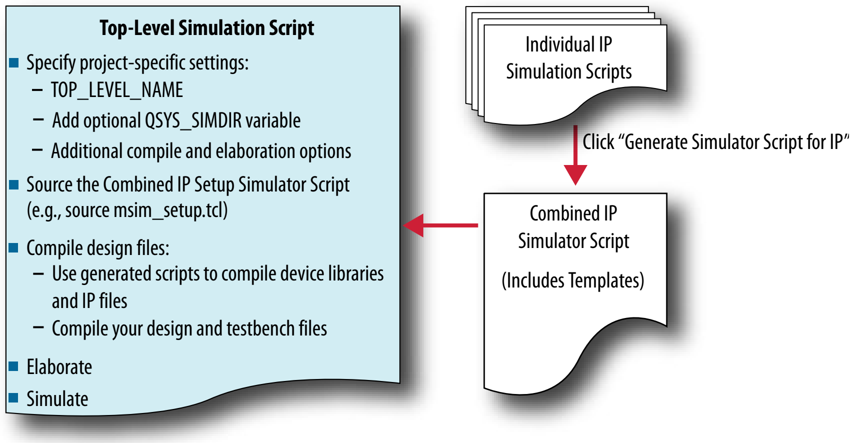
AN 343: Intel FPGA IP Evaluation Mode of AMPP IP

1.6.2 Scripting IP Simulation

The Intel Quartus Prime software supports the use of scripts to automate simulation processing in your preferred simulation environment. Use the scripting methodology that you prefer to control simulation.

Use a version-independent, top-level simulation script to control design, testbench, and IP core simulation. Because Intel Quartus Prime-generated simulation file names may change after IP upgrade or regeneration, your top-level simulation script must "source" the generated setup scripts, rather than using the generated setup scripts directly. Follow these steps to generate or regenerate combined simulator setup scripts:

Figure 1. Incorporating Generated Simulator Setup Scripts into a Top-Level Simulation Script


Top-Level Simulation Script
Specify project-specific settings:
– TOP_LEVEL_NAME
– Add optional QSYS_SIMDIR variable
– Additional compile and elaboration options
Source the Combined IP Setup Simulator Script (e.g., source msim_setup.tcl)
Compile design files:
– Use generated scripts to compile device libraries and IP files
– Compile your design and testbench files
Elaborate
Simulate
Individual IP Simulation Scripts
Click "Generate Simulator Script for IP"
Combined IP Simulator Script
(Includes Templates)

1. Click **Project ➤ Upgrade IP Components ➤ Generate Simulator Script for IP** (or run the `ip-setup-simulation` utility) to generate or regenerate a combined simulator setup script for all IP for each simulator.
2. Use the templates in the generated script to source the combined script in your top-level simulation script. Each simulator's combined script file contains a rudimentary template that you adapt for integration of the setup script into a top-level simulation script.

   This technique eliminates manual update of simulation scripts if you modify or upgrade the IP variation.

1.6.2.1 Generating a Combined Simulator Setup Script

Run the **Generate Simulator Setup Script for IP** command to generate a combined simulator setup script.

Source this combined script from a top-level simulation script. Click **Tools ➤ Generate Simulator Setup Script for IP** (or use of the `ip-setup-simulation` utility at the command-line) to generate or update the combined scripts, after any of the following occur:

- IP core initial generation or regeneration with new parameters
- Intel Quartus Prime software version upgrade
- IP core version upgrade

To generate a combined simulator setup script for all project IP cores for each simulator:

1. Generate, regenerate, or upgrade one or more IP core. Refer to *Generating IP Cores* or *Upgrading IP Cores*.
2. Click **Tools ➤ Generate Simulator Setup Script for IP** (or run the `ip-setup-simulation` utility). Specify the **Output Directory** and library compilation options. Click **OK** to generate the file. By default, the files generate into the */`<project directory>`/`<simulator>`/* directory using relative paths.
3. To incorporate the generated simulator setup script into your top-level simulation script, refer to the template section in the generated simulator setup script as a guide to creating a top-level script:
   a. Copy the specified template sections from the simulator-specific generated scripts and paste them into a new top-level file.
   b. Remove the comments at the beginning of each line from the copied template sections.
   c. Specify the customizations you require to match your design simulation requirements, for example:
      - Specify the TOP_LEVEL_NAME variable to the design's simulation top-level file. The top-level entity of your simulation is often a testbench that instantiates your design. Then, your design instantiates IP cores or Platform Designer systems. Set the value of TOP_LEVEL_NAME to the top-level entity.
      - If necessary, set the `QSYS_SIMDIR` variable to point to the location of the generated IP simulation files.
      - Compile the top-level HDL file (for example, a test program) and all other files in the design.
      - Specify any other changes, such as using the `grep` command-line utility to search a transcript file for error signatures, or e-mail a report.
4. Re-run **Tools ➤ Generate Simulator Setup Script for IP** (or `ip-setup-simulation`) after regeneration of an IP variation.

Table 7. Simulation Script Utilities

Utility	Syntax
`ip-setup-simulation` generates a combined, version-independent simulation script for all Intel FPGA IP cores in your project. The command also automates regeneration of the script after upgrading software or IP versions. Use the `compile-to-work` option to	`ip-setup-simulation --quartus-project=<my proj> --output-directory=<my_dir> --use-relative-paths --compile-to-work`

continued...

Utility	Syntax
compile all simulation files into a single work library if your simulation environment requires. Use the `--use-relative-paths` option to use relative paths whenever possible.	`--use-relative-paths` and `--compile-to-work` are optional. For command-line help listing all options for these executables, type: `<utility name>` --help.
`ip-make-simscript` generates a combined simulation script for all IP cores that you specify on the command line. Specify one or more `.spd` files and an output directory in the command. Running the script compiles IP simulation models into various simulation libraries.	`ip-make-simscript --spd=<ipA.spd,ipB.spd> --output-directory=<directory>`

The following sections provide step-by-step instructions for sourcing each simulator setup script in your top-level simulation script.

1.6.2.2 Incorporating Simulator Setup Scripts from the Generated Template

You can incorporate generated IP core simulation scripts into a top-level simulation script that controls simulation of your entire design. After running `ip-setup-simulation` use the following information to copy the template sections and modify them for use in a new top-level script file.

1.6.2.2.1 Sourcing Aldec* Simulator Setup Scripts

Follow these steps to incorporate the generated Aldec simulation scripts into a top-level project simulation script.

1. The generated simulation script contains the following template lines. Cut and paste these lines into a new file. For example, `sim_top.tcl`.

```
# # Start of template
# # If the copied and modified template file is "aldec.do", run it as:
# # vsim -c -do aldec.do
# #
# # Source the generated sim script
# source rivierapro_setup.tcl
# # Compile eda/sim_lib contents first
# dev_com
# # Override the top-level name (so that elab is useful)
# set TOP_LEVEL_NAME top
# # Compile the standalone IP.
# com
# # Compile the top-level
# vlog -sv2k5 ../../top.sv
# # Elaborate the design.
# elab
# # Run the simulation
# run
# # Report success to the shell
# exit -code 0
# # End of template
```

2. Delete the first two characters of each line (comment and space):

```
# Start of template
# If the copied and modified template file is "aldec.do", run it as:
# vsim -c -do aldec.do
#
# Source the generated sim script source rivierapro_setup.tcl
# Compile eda/sim_lib contents first dev_com
# Override the top-level name (so that elab is useful)
set TOP_LEVEL_NAME top
# Compile the standalone IP.
```

```
com
# Compile the top-level vlog -sv2k5 ../../top.sv
# Elaborate the design.
elab
# Run the simulation
run
# Report success to the shell
exit -code 0
# End of template
```

3. Modify the TOP_LEVEL_NAME and compilation step appropriately, depending on the simulation's top-level file. For example:

```
set TOP_LEVEL_NAME sim_top
 vlog –sv2k5 ../../sim_top.sv
```

4. If necessary, add the QSYS_SIMDIR variable to point to the location of the generated IP simulation files. Specify any other changes that you require to match your design simulation requirements. The scripts offer variables to set compilation or simulation options. Refer to the generated script for details.
5. Run the new top-level script from the generated simulation directory:

```
vsim –c –do <path to sim_top>.tcl
```

1.6.2.2.2 Sourcing Cadence* Simulator Setup Scripts

Follow these steps to incorporate the generated Cadence IP simulation scripts into a top-level project simulation script.

1. The generated simulation script contains the following template lines. Cut and paste these lines into a new file. For example, ncsim.sh.

```
# # Start of template
# # If the copied and modified template file is "ncsim.sh", run it as:
# # ./ncsim.sh
# #
# # Do the file copy, dev_com and com steps
# source ncsim_setup.sh \
# SKIP_ELAB=1 \
# SKIP_SIM=1
#
# # Compile the top level module
# ncvlog -sv "$QSYS_SIMDIR/../top.sv"
#
# # Do the elaboration and sim steps
# # Override the top-level name
# # Override the sim options, so the simulation
# # runs forever (until $finish()).
# source ncsim_setup.sh \
# SKIP_FILE_COPY=1 \
# SKIP_DEV_COM=1 \
# SKIP_COM=1 \
# TOP_LEVEL_NAME=top \
# USER_DEFINED_SIM_OPTIONS=""
# # End of template
```

2. Delete the first two characters of each line (comment and space):

```
# Start of template
# If the copied and modified template file is "ncsim.sh", run it as:
# ./ncsim.sh
#
# Do the file copy, dev_com and com steps
source ncsim_setup.sh \
SKIP_ELAB=1 \
SKIP_SIM=1
```

```
# Compile the top level module
ncvlog -sv "$QSYS_SIMDIR/../top.sv"
# Do the elaboration and sim steps
# Override the top-level name
# Override the sim options, so the simulation
# runs forever (until $finish()).
source ncsim_setup.sh \
SKIP_FILE_COPY=1 \
SKIP_DEV_COM=1 \
SKIP_COM=1 \
TOP_LEVEL_NAME=top \
USER_DEFINED_SIM_OPTIONS=""
# End of template
```

3. Modify the `TOP_LEVEL_NAME` and compilation step appropriately, depending on the simulation's top-level file. For example:

```
TOP_LEVEL_NAME=sim_top \
 ncvlog -sv "$QSYS_SIMDIR/../top.sv"
```

4. If necessary, add the `QSYS_SIMDIR` variable to point to the location of the generated IP simulation files. Specify any other changes that you require to match your design simulation requirements. The scripts offer variables to set compilation or simulation options. Refer to the generated script for details.

5. Run the resulting top-level script from the generated simulation directory by specifying the path to `ncsim.sh`.

1.6.2.2.3 Sourcing ModelSim* Simulator Setup Scripts

Follow these steps to incorporate the generated ModelSim IP simulation scripts into a top-level project simulation script.

1. The generated simulation script contains the following template lines. Cut and paste these lines into a new file. For example, `sim_top.tcl`.

```
# # Start of template
# # If the copied and modified template file is "mentor.do", run it
# # as: vsim -c -do mentor.do
# #
# # Source the generated sim script
# source msim_setup.tcl
# # Compile eda/sim_lib contents first
# dev_com
# # Override the top-level name (so that elab is useful)
# set TOP_LEVEL_NAME top
# # Compile the standalone IP.
# com
# # Compile the top-level
# vlog -sv ../../top.sv
# # Elaborate the design.
# elab
# # Run the simulation
# run -a
# # Report success to the shell
# exit -code 0
# # End of template
```

2. Delete the first two characters of each line (comment and space):

```
# Start of template
# If the copied and modified template file is "mentor.do", run it
# as: vsim -c -do mentor.do
#
# Source the generated sim script source msim_setup.tcl
# Compile eda/sim_lib contents first
dev_com
```

```
# Override the top-level name (so that elab is useful)
set TOP_LEVEL_NAME top
# Compile the standalone IP.
com
# Compile the top-level vlog -sv ../../top.sv
# Elaborate the design.
elab
# Run the simulation
run -a
# Report success to the shell
exit -code 0
# End of template
```

3. Modify the `TOP_LEVEL_NAME` and compilation step appropriately, depending on the simulation's top-level file. For example:

```
set TOP_LEVEL_NAME sim_top vlog -sv ../../sim_top.sv
```

4. If necessary, add the `QSYS_SIMDIR` variable to point to the location of the generated IP simulation files. Specify any other changes required to match your design simulation requirements. The scripts offer variables to set compilation or simulation options. Refer to the generated script for details.
5. Run the resulting top-level script from the generated simulation directory:

```
vsim –c –do <path to sim_top>.tcl
```

1.6.2.2.4 Sourcing VCS* Simulator Setup Scripts

Follow these steps to incorporate the generated Synopsys VCS simulation scripts into a top-level project simulation script.

1. The generated simulation script contains these template lines. Cut and paste the lines preceding the "helper file" into a new executable file. For example, `synopsys_vcs.f`.

```
# # Start of template
# # If the copied and modified template file is "vcs_sim.sh", run it
# # as: ./vcs_sim.sh
# #
# # Override the top-level name
# # specify a command file containing elaboration options
# # (system verilog extension, and compile the top-level).
# # Override the sim options, so the simulation
# # runs forever (until $finish()).
# source vcs_setup.sh \
# TOP_LEVEL_NAME=top \
# USER_DEFINED_ELAB_OPTIONS="'-f ../../../synopsys_vcs.f'" \
# USER_DEFINED_SIM_OPTIONS=""
#
# # helper file: synopsys_vcs.f
# +systemverilogext+.sv
# ../../../top.sv
# # End of template
```

2. Delete the first two characters of each line (comment and space) for the `vcs.sh` file, as shown below:

```
# Start of template
# If the copied and modified template file is "vcs_sim.sh", run it
# as: ./vcs_sim.sh
#
# Override the top-level name
# specify a command file containing elaboration options
# (system verilog extension, and compile the top-level).
# Override the sim options, so the simulation
# runs forever (until $finish()).
source vcs_setup.sh \
TOP_LEVEL_NAME=top \
USER_DEFINED_ELAB_OPTIONS="'-f ../../../synopsys_vcs.f'" \
USER_DEFINED_SIM_OPTIONS=""
```

3. Delete the first two characters of each line (comment and space) for the `synopsys_vcs.f` file, as shown below:

```
# helper file: synopsys_vcs.f
 +systemverilogext+.sv
 ../../../top.sv
# End of template
```

4. Modify the `TOP_LEVEL_NAME` and compilation step appropriately, depending on the simulation's top-level file. For example:

```
TOP_LEVEL_NAME=sim_top \
```

5. If necessary, add the `QSYS_SIMDIR` variable to point to the location of the generated IP simulation files. Specify any other changes required to match your design simulation requirements. The scripts offer variables to set compilation or simulation options. Refer to the generated script for details.
6. Run the resulting top-level script from the generated simulation directory by specifying the path to `vcs_sim.sh`.

1.6.2.2.5 Sourcing VCS* MX Simulator Setup Scripts

Follow these steps to incorporate the generated Synopsys VCS MX simulation scripts for use in top-level project simulation scripts.

1. The generated simulation script contains these template lines. Cut and paste the lines preceding the "helper file" into a new executable file. For example, `vcsmx.sh`.

```
# # Start of template
# # If the copied and modified template file is "vcsmx_sim.sh", run
# # it as: ./vcsmx_sim.sh
# #
# # Do the file copy, dev_com and com steps
# source vcsmx_setup.sh \
# SKIP_ELAB=1 \

# SKIP_SIM=1
#
# # Compile the top level module vlogan +v2k
     +systemverilogext+.sv "$QSYS_SIMDIR/../top.sv"

# # Do the elaboration and sim steps
# # Override the top-level name
# # Override the sim options, so the simulation runs
# # forever (until $finish()).
# source vcsmx_setup.sh \
# SKIP_FILE_COPY=1 \
# SKIP_DEV_COM=1 \
# SKIP_COM=1 \
# TOP_LEVEL_NAME="'-top top'" \
# USER_DEFINED_SIM_OPTIONS=""
# # End of template
```

2. Delete the first two characters of each line (comment and space), as shown below:

```
# Start of template
# If the copied and modified template file is "vcsmx_sim.sh", run
# it as: ./vcsmx_sim.sh
#
# Do the file copy, dev_com and com steps
source vcsmx_setup.sh \
SKIP_ELAB=1 \
SKIP_SIM=1

# Compile the top level module
vlogan +v2k +systemverilogext+.sv "$QSYS_SIMDIR/../top.sv"

# Do the elaboration and sim steps
# Override the top-level name
# Override the sim options, so the simulation runs
# forever (until $finish()).
source vcsmx_setup.sh \
SKIP_FILE_COPY=1 \
SKIP_DEV_COM=1 \
SKIP_COM=1 \
TOP_LEVEL_NAME="'-top top'" \
USER_DEFINED_SIM_OPTIONS=""
# End of template
```

3. Modify the TOP_LEVEL_NAME and compilation step appropriately, depending on the simulation's top-level file. For example:

```
TOP_LEVEL_NAME=”-top sim_top’” \
```

4. Make the appropriate changes to the compilation of the your top-level file, for example:

```
vlogan +v2k +systemverilogext+.sv "$QSYS_SIMDIR/../sim_top.sv"
```

5. If necessary, add the QSYS_SIMDIR variable to point to the location of the generated IP simulation files. Specify any other changes required to match your design simulation requirements. The scripts offer variables to set compilation or simulation options. Refer to the generated script for details.
6. Run the resulting top-level script from the generated simulation directory by specifying the path to vcsmx_sim.sh.

1.7 Using NativeLink Simulation (Intel Quartus Prime Standard Edition)

The NativeLink feature integrates your EDA simulator with the Intel Quartus Prime Standard Edition software by automating the following:

- Generation of simulator-specific files and simulation scripts.
- Compilation of simulation libraries.
- Launches your simulator automatically following Intel Quartus Prime Analysis & Elaboration, Analysis & Synthesis, or after a full compilation.

Note: The Intel Quartus Prime Pro Edition does not support NativeLink simulation. If you use NativeLink for Intel Arria 10 devices in the Intel Quartus Prime Standard Edition, you must add the `.qsys` file generated for the IP or Platform Designer (Standard) system to your Intel Quartus Prime project. If you use NativeLink for any other supported device family, you must add the `.qip` and `.sip` files to your project.

1.7.1 Setting Up NativeLink Simulation (Intel Quartus Prime Standard Edition)

Before running NativeLink simulation, specify settings for your simulator in the Intel Quartus Prime software.

To specify NativeLink settings in the Intel Quartus Prime Standard Edition software, follow these steps:

1. Open an Intel Quartus Prime Standard Edition project.
2. Click **Tools > Options** and specify the location of your simulator executable file.

Table 8. Execution Paths for EDA Simulators

Simulator	Path
Mentor Graphics ModelSim-AE	`<drive letter>:\<simulator install path>\win32aloem` (Windows) `/<simulator install path>/bin` (Linux)
Mentor Graphics ModelSim Mentor Graphics QuestaSim	`<drive letter>:\<simulator install path>\win32` (Windows) `<simulator install path>/bin` (Linux)
Synopsys VCS/VCS MX	`<simulator install path>/bin` (Linux)
Cadence Incisive Enterprise	`<simulator install path>/tools/bin` (Linux)
Aldec Active-HDL Aldec Riviera-PRO	`<drive letter>:\<simulator install path>\bin` (Windows) `<simulator install path>/bin` (Linux)

3. Click **Assignments ➤ Settings** and specify options on the **Simulation** page and the **More NativeLink Settings** dialog box. Specify default options for simulation library compilation, netlist and tool command script generation, and for launching RTL or gate-level simulation automatically following compilation.
4. If your design includes a testbench, turn on **Compile test bench**. Click **Test Benches** to specify options for each testbench. Alternatively, turn on **Use script to compile testbench** and specify the script file.
5. To use a script to setup a simulation, turn on **Use script to setup simulation**.

1.7.2 Running RTL Simulation (NativeLink Flow)

To run RTL simulation using the NativeLink flow, follow these steps:

1. Set up the simulation environment.
2. Click **Processing > Start > Analysis and Elaboration**.
3. Click **Tools > Run Simulation Tool > RTL Simulation**.

   NativeLink compiles simulation libraries and launches and runs your RTL simulator automatically according to the NativeLink settings.
4. Review and analyze the simulation results in your simulator. Correct any functional errors in your design. If necessary, re-simulate the design to verify correct behavior.

1.7.3 Running Gate-Level Simulation (NativeLink Flow)

To run gate-level simulation with the NativeLink flow, follow these steps:

1. Prepare for simulation.
2. Set up the simulation environment. To generate only a functional (rather than timing) gate-level netlist, click **More EDA Netlist Writer Settings**, and turn on **Generate netlist for functional simulation only**.
3. To synthesize the design, follow one of these steps:
   - To generate a post-fit functional or post-fit timing netlist and then automatically simulate your design according to your NativeLink settings, Click **Processing > Start Compilation**. Skip to step 6.
   - To synthesize the design for post-synthesis functional simulation only, click **Processing > Start > Start Analysis and Synthesis**.
4. To generate the simulation netlist, click **Start EDA Netlist Writer**.
5. Click **Tools > Run Simulation Tool > Gate Level Simulation**.
6. Review and analyze the simulation results in your simulator. Correct any unexpected or incorrect conditions found in your design. Simulate the design again until you verify correct behavior.

1.8 Running a Simulation (Custom Flow)

Use a custom simulation flow to support any of the following more complex simulation scenarios:

- Custom compilation, elaboration, or run commands for your design, IP, or simulation library model files (for example, macros, debugging/optimization options, simulator-specific elaboration or run-time options)
- Multi-pass simulation flows
- Flows that use dynamically generated simulation scripts

Use these to compile libraries and generate simulation scripts for custom simulation flows:

- NativeLink-generated scripts—use NativeLink only to generate simulation script templates to develop your own custom scripts.
- Simulation Library Compiler—compile Intel FPGA simulation libraries for your device, HDL, and simulator. Generate scripts to compile simulation libraries as part of your custom simulation flow. This tool does not compile your design, IP, or testbench files.
- IP and Platform Designer (Standard) simulation scripts—use the scripts generated for Intel FPGA IP cores and Platform Designer (Standard) systems as templates to create simulation scripts. If your design includes multiple IP cores or Platform Designer (Standard) systems, you can combine the simulation scripts into a single script, manually or by using the `ip-make-simscript` utility.

Use the following steps in a custom simulation flow:

1. Compile the design and testbench files in your simulator.
2. Run the simulation in your simulator.

Post-synthesis and post-fit gate-level simulations run significantly slower than RTL simulation. Intel FPGA recommends that you verify your design using RTL simulation for functionality and use the Timing Analyzer for timing. Timing simulation is not supported for Arria V, Cyclone V, Stratix V, and newer families.

1.9 Document Revision History

This document has the following revision history.

Date	Version	Changes
2017.11.06	17.1.0	• Added Simulation Library Compiler details to Quick Start Example
2017.05.08	17.0.0	• Gate-level timing simulation limited to Arria II GX/GZ,Cyclone IV, MAX II, MAX V, and Stratix IV device families.
2016.10.31	16.1.0	• Updated simulator support table with latest version information. • Clarified license requirements for mixed language simulation with VHDL. • Gate-level timing simulation limited to Stratix IV and Cyclone IV devices.
2016.05.02	16.0.0	• Noted limitations of NativeLink simulation. • Updated simulator support table with latest version information.
2015.11.02	15.1.0	• Added new Generating Version-Independent IP Simulation Scripts topic. • Added example IP simulation script templates for all supported simulators. • Added new Incorporating IP Simulation Scripts in Top-Level Scripts topic. • Updated simulator support table with latest version information. • Changed instances of *Quartus II* to *Quartus Prime*.
2015.05.04	15.0.0	• Updated simulator support table with latest. • Gate-level timing simulation limited to Stratix IV and Cyclone IV devices. • Added mixed language simulation support in the ModelSim - Intel FPGA Edition software.
2014.06.30	14.0.0	• Replaced MegaWizard Plug-In Manager information with IP Catalog.
May 2013	13.0.0	• Updated introductory section and system and IP file locations.

continued...

Date	Version	Changes
November 2012	12.1.0	• Revised chapter to reflect latest changes to other simulation documentation.
June 2012	12.0.0	• Reorganization of chapter to reflect various simulation flows. • Added NativeLink support for newer IP cores.
November 2011	11.1.0	• Added information about encrypted Altera simulation model files. • Added information about IP simulation and NativeLink.

Related Links

Documentation Archive
For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

2 ModelSim - Intel FPGA Edition, ModelSim, and QuestaSim Support*

You can include your supported EDA simulator in the Intel Quartus Prime design flow. This document provides guidelines for simulation of designs with ModelSim or QuestaSim software. The entry-level ModelSim - Intel FPGA Edition includes precompiled simulation libraries.

Note: The latest version of theModelSim - Intel FPGA Edition software supports native, mixed-language (VHDL/Verilog HDL/SystemVerilog) co-simulation of plain text HDL. If you have a VHDL-only simulator, you can use the ModelSim-Intel FPGA Edition software to simulate Verilog HDL modules and IP cores. Alternatively, you can purchase separate co-simulation software.

Related Links

- Simulating Designs on page 11
- Managing Intel Quartus Prime Projects

2.1 Quick Start Example (ModelSim with Verilog)

You can adapt the following RTL simulation example to get started quickly with ModelSim:

1. To specify your EDA simulator and executable path, type the following Tcl package command in the Intel Quartus Prime tcl shell window:

   `set_user_option -name EDA_TOOL_PATH_MODELSIM` *<modelsim executable path>*

   `set_global_assignment -name EDA_SIMULATION_TOOL "MODELSIM (verilog)"`

2. Compile simulation model libraries using one of the following methods:

   - Run NativeLink RTL simulation to compile required design files, simulation models, and run your simulator. Verify results in your simulator. If you complete this step you can ignore the remaining steps.
   - To automatically compile all required simulation model libraries for your design in your supported simulator, click **Tools ➤ Launch Simulation Library Compiler**. Specify options for your simulation tool, language, target device family, and output location, and then click **OK**.
   - Type the following commands to create and map Intel FPGA simulation libraries manually, and then compile the models manually:

     ```
     vlib <lib1>_ver
     vmap <lib1>_ver <lib1>_ver
     vlog -work <lib1> <lib1>
     ```

Use the compiled simulation model libraries during simulatation of your design. Refer to your EDA simulator's documentation for information about running simulation.

3. Compile your design and testbench files:

```
vlog -work work <design or testbench name>.v
```

4. Load the design:

```
vsim -L work -L <lib1>_ver -L <lib2>_ver work.<testbench name>
```

2.2 ModelSim, ModelSim-Intel FPGA Edition, and QuestaSim Guidelines

The following guidelines apply to simulation of designs in the ModelSim, ModelSim-Intel FPGA Edition, or QuestaSim software.

2.2.1 Using ModelSim-Intel FPGA Edition Precompiled Libraries

Precompiled libraries for both functional and gate-level simulations are provided for the ModelSim-Intel FPGA Edition software. You should not compile these library files before running a simulation. No precompiled libraries are provided for ModelSim or QuestaSim. You must compile the necessary libraries to perform functional or gate-level simulation with these tools.

The precompiled libraries provided in <*install path*>`/altera/` must be compatible with the version of the Intel Quartus Prime software that creates the simulation netlist. To verify compatibility of precompiled libraries with your version of the Intel Quartus Prime software, refer to the <*install path*>`/altera/version.txt` file. This file indicates the Intel Quartus Prime software version and build of the precompiled libraries.

Note: Encrypted simulation model files shipped with the Intel Quartus Prime software version 10.1 and later can only be read by ModelSim-Intel FPGA Edition software version 6.6c and later. These encrypted simulation model files are located at the <*Intel Quartus Prime System directory*>`/quartus/eda/sim_lib/`<*mentor*> directory.

2.2.2 Disabling Timing Violation on Registers

In certain situations, you may want to ignore timing violations on registers and disable the “X” propagation that occurs. For example, this technique may be helpful to eliminate timing violations in internal synchronization registers in asynchronous clock-domain crossing. Intel Arria 10 devices do not support timing simulation. Intel Arria 10 devices do not support timing simulation.

By default, the **x_on_violation_option** logic option is enabled for all design registers, resulting in an output of “X” at timing violation. To disable “X” propagation at timing violations on a specific register, disable the **x_on_violation_option** logic option for the specific register, as shown in the following example from the Intel Quartus Prime Settings File (`.qsf`).

```
set_instance_assignment -name X_ON_VIOLATION_OPTION OFF -to \
<register_name>
```

2.2.3 Passing Parameter Information from Verilog HDL to VHDL

You must use in-line parameters to pass values from Verilog HDL to VHDL.

By default, the **x_on_violation_option** logic option is enabled for all design registers, resulting in an output of "X" at timing violation. To disable "X" propagation at timing violations on a specific register, disable the **x_on_violation_option** logic option for the specific register, as shown in the following example from the Intel Quartus Prime Settings File (`.qsf`).

```
set_instance_assignment -name X_ON_VIOLATION_OPTION OFF -to \
<register_name>
```

Example 1. In-line Parameter Passing Example

```
lpm_add_sub#(.lpm_width(12), .lpm_direction("Add"),
.lpm_type("LPM_ADD_SUB"),
.lpm_hint("ONE_INPUT_IS_CONSTANT=NO,CIN_USED=NO" ))

lpm_add_sub_component (
        .dataa (dataa),
        .datab (datab),
        .result (sub_wire0)
);
```

Note: The sequence of the parameters depends on the sequence of the GENERIC in the VHDL component declaration.

2.2.4 Increasing Simulation Speed

By default, the ModelSim and QuestaSim software runs in a debug-optimized mode.

To run the ModelSim and QuestaSim software in speed-optimized mode, add the following two vlog command-line switches. In this mode, module boundaries are flattened and loops are optimized, which eliminates levels of debugging hierarchy and may result in faster simulation. This switch is not supported in the ModelSim-Intel FPGA Edition simulator.

```
vlog -fast -05
```

2.2.5 Simulating Transport Delays

By default, the ModelSim and QuestaSim software filter out all pulses that are shorter than the propagation delay between primitives.

Turning on the **transport delay** options in the ModelSim and QuestaSim software prevents the simulator from filtering out these pulses. Intel Arria 10 devices do not support timing simulation.

Table 9. Transport Delay Simulation Options (ModelSim and QuestaSim)

Option	Description
`+transport_path_delays`	Use when simulation pulses are shorter than the delay in a gate-level primitive. You must include the `+pulse_e/number` and `+pulse_r/number` options.
`+transport_int_delays`	Use when simulation pulses are shorter than the interconnect delay between gate-level primitives. You must include the `+pulse_int_e/number` and `+pulse_int_r/number` options.

Note: The `+transport_path_delays` and `+transport_path_delays` options apply automatically during NativeLink gate-level timing simulation. For more information about either of these options, refer to the ModelSim-Intel FPGA Edition Command Reference installed with the ModelSim and QuestaSim software.

The following ModelSim and QuestaSim software command shows the command line syntax to perform a gate-level timing simulation with the device family library:

```
vsim -t 1ps -L stratixii -sdftyp /i1=filtref_vhd.sdo work.filtref_vhd_vec_tst
\
+transport_int_delays +transport_path_delays
```

2.2.6 Viewing Simulation Messages

ModelSim and QuestaSim error and warning messages are tagged with a vsim or vcom code. To determine the cause and resolution for a vsim or vcom error or warning, use the verror command.

For example, ModelSim may return the following error:

```
# ** Error: C:/altera_trn/DUALPORT_TRY/simulation/modelsim/
DUALPORT_TRY.vho(31):
 (vcom-1136) Unknown identifier "stratixiv"
```

In this case, type the following command:

```
verror 1136
```

The following description appears:

```
# vcom Message # 1136:
# The specified name was referenced but was not found. This indicates
# that either the name specified does not exist or is not visible at
# this point in the code.
```

Note: If your design includes deep levels of hierarchy, and the **Maintain hierarchy** EDA tools option is turned on, this may result in a large number of module instances in post-fit or post-map netlist. This condition can exceed the ModelSim-Intel FPGA Edition instance limitation.

To avoid exceeding any ModelSim-Intel FPGA Edition instance limit, turn off **Maintain hierarchy** to reduce the number of modules instances to 1 in the post-fit or post-map netlist. To acces this option, click **Assignments ➤ Settings ➤ EDA Tool Settings ➤ More Settings**.

2.2.7 Generating Power Analysis Files

To generate a timing Value Change Dump File (`.vcd`) for power analysis, you must first generate a `<`*`filename`*`>_dump_all_vcd_nodes.tcl` script file in the Intel Quartus Prime software. You can then run the script from the ModelSim, QuestaSim, or ModelSim-Intel FPGA Edition software to generate a timing `.vcd` for use in the Intel Quartus Prime power analyzer.

To generate and use a `.vcd` for power analysis, follow these steps:

1. In the Intel Quartus Prime software, click **Assignments ➤ Settings**.
2. Under **EDA Tool Settings**, click **Simulation**.
3. Turn on **Generate Value Change Dump file script**, specify the type of output signals to include, and specify the top-level design instance name in your testbench. For example, if your top level design name is Top, and your testbench wrapper calls Top as instance `Top_inst`, specify the top level design instance name as `Top_inst`.
4. Click **Processing ➤ Start Compilation**. The Compiler creates the `<`*`filename`*`>_dump_all_vcd_nodes.tcl` file, the ModelSim simulation `<`*`filename`*`>_run_msim_gate_vhdl/verilog`.do file (including the **.vcd** and **.tcl** execution lines). Use the `<`*`filename`*`>_dump_all_vcd_nodes.tcl` to dump all of the signals that you expect for input back into the Power Analysis.
5. Elaborate and compile the design in your simulator.
6. Source the `<`*`filename`*`>_run_msim_gate_vhdl/verilog`.do file, and then run the simulation. The simulator opens the `.vcd` file that contains the dumped signal file transition information.
7. Stop the simulation if your testbench does not have a break point.

2.2.8 Viewing Simulation Waveforms

ModelSim-Intel FPGA Edition, ModelSim, and QuestaSim automatically generate a Wave Log Format File (`.wlf`) following simulation. You can use the `.wlf` to generate a waveform view.

To view a waveform from a `.wlf` through ModelSim-Intel FPGA Edition, ModelSim, or QuestaSim, perform the following steps:

1. Type `vsim` at the command line. The **ModelSim/QuestaSim** or **ModelSim-Intel FPGA Edition** dialog box appears.
2. Click **File ➤ Datasets**. The **Datasets Browser** dialog box appears.
3. Click **Open** and select your `.wlf`.
4. Click **Done**.
5. In the Object browser, select the signals that you want to observe.
6. Click **Add ➤ Wave**, and then click **Selected Signals**.

You must first convert the .vcd to a .wlf before you can view a waveform in ModelSim-Intel FPGA Edition, ModelSim, or QuestaSim.

7. To convert the the .vcd to a .wlf, type the following at the command-line:

```
vcd2wlf <example>.vcd <example>.wlf
```

8. After conversion, view the .wlf waveform in ModelSim or QuestaSim.

2.2.9 Simulating with ModelSim-Intel FPGA Edition Waveform Editor

You can use the ModelSim-Intel FPGA Edition waveform editor as a simple method to create stimulus vectors for simulation. You can create this design stimulus via interactive manipulation of waveforms from the wave window in ModelSim-Intel FPGA Edition. With the ModelSim-Intel FPGA Edition waveform editor, you can create and edit waveforms, drive simulation directly from created waveforms, and save created waveforms into a stimulus file.

Related Links

ModelSim Web Page

2.3 ModelSim Simulation Setup Script Example

The Intel Quartus Prime software can generate a msim_setup.tcl simulation setup script for IP cores in your design. The script compiles the required device library models, compiles the design files, and elaborates the design with or without simulator optimization. To run the script, type source `msim_setup.tcl` in the simulator Transcript window.

Alternatively, if you are using the simulator at the command line, you can type the following command:

```
vsim -c -do msim_setup.tcl
```

In this example the `top-level-simulate.do` custom top-level simulation script sets the hierarchy variable TOP_LEVEL_NAME to `top_testbench` for the design, and sets the variable QSYS_SIMDIR to the location of the generated simulation files.

```
# Set hierarchy variables used in the IP-generated files
set TOP_LEVEL_NAME "top_testbench"
set QSYS_SIMDIR "./ip_top_sim"
# Source generated simulation script which defines aliases used below
source $QSYS_SIMDIR/mentor/msim_setup.tcl
# dev_com alias compiles simulation libraries for device library files
dev_com
# com alias compiles IP simulation or Qsys model files and/or Qsys model
files in the correct order
com
# Compile top level testbench that instantiates your IP
vlog -sv ./top_testbench.sv
# elab alias elaborates the top-level design and testbench
elab
# Run the full simulation
run - all
```

In this example, the top-level simulation files are stored in the same directory as the original IP core, so this variable is set to the IP-generated directory structure. The QSYS_SIMDIR variable provides the relative hierarchy path for the generated IP simulation files. The script calls the generated `msim_setup.tcl` script and uses the alias commands from the script to compile and elaborate the IP files required for simulation along with the top-level simulation testbench. You can specify additional simulator elaboration command options when you run the `elab` command, for example, `elab +nowarnTFMPC`. The last command run in the example starts the simulation.

2.4 Unsupported Features

The Intel Quartus Prime software does not support the following ModelSim simulation features:

- Intel Quartus Prime does not support companion licensing for ModelSim.
- The USB software guard is not supported by versions earlier than ModelSim software version 5.8d.
- For ModelSim software versions prior to 5.5b, use the **PCLS** utility included with the software to set up the license.
- Some versions of ModelSim and QuestaSim support SystemVerilog, PSL assertions, SystemC, and more. For more information about specific feature support, refer to Mentor Graphics literature

Related Links

ModelSim-Intel FPGA Edition Software Web Page

2.5 Document Revision History

Table 10. Document Revision History

Date	Version	Changes
2017.11.06	17.1.0	• Changed title to ModelSim - Intel FPGA Edition, ModelSim, and QuestaSim Support* • Stated no support for Intel Arria 10 timing simulation in Simulating Transport Delays and Disabling Timing Violations on Registers topics. • Added Simulation Library Compiler details and another step to Quick Start Example
2016.10.31	16.1.0	• Implemented Intel rebranding. • Corrected load design syntax error.
2016.05.02	16.0.0	• Noted limitations of NativeLink simulation. • Added note about avoiding ModelSim - Intel FPGA Edition instance limitations.
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Quartus Prime*.
2015.05.04	15.0.0	• Added mixed language simulation support in the ModelSim - Intel FPGA Edition software.
2014.06.30	14.0.0	• Replaced MegaWizard Plug-In Manager information with IP Catalog.

continued...

Date	Version	Changes
November 2012	12.1.0	• Relocated general simulation information to Simulating Altera Designs.
June 2012	12.0.0	• Removed survey link.
November 2011	11.0.1	• Changed to new document template.

Related Links

Documentation Archive
For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

3 Synopsys VCS and VCS MX Support

You can include your supported EDA simulator in the Intel Quartus Prime design flow. This document provides guidelines for simulation of Intel Quartus Prime designs with the Synopsys VCS or VCS MX software.

3.1 Quick Start Example (VCS with Verilog)

You can adapt the following RTL simulation example to get started quickly with VCS:

1. To specify your EDA simulator and executable path, type the following Tcl package command in the Intel Quartus Prime tcl shell window:

   `set_user_option -name EDA_TOOL_PATH_VCS` <*VCS executable path*>

   `set_global_assignment -name EDA_SIMULATION_TOOL "VCS"`

2. Compile simulation model libraries using one of the following methods:

   - Run NativeLink RTL simulation to compile required design files, simulation models, and run your simulator. Verify results in your simulator. If you complete this step you can ignore the remaining steps.
   - To automatically compile all required simulation model libraries for your design in your supported simulator, click **Tools ➤ Launch Simulation Library Compiler**. Specify options for your simulation tool, language, target device family, and output location, and then click **OK**.

   Use the compiled simulation model libraries during simulatation of your design. Refer to your EDA simulator's documentation for information about running simulation.

3. Modify the `simlib_comp.vcs` file to specify your design and testbench files.
4. Type the following to run the VCS simulator:

   ```
   vcs -R -file simlib_comp.vcs
   ```

3.2 VCS and QuestaSim Guidelines

The following guidelines apply to simulation of Intel FPGA designs in the VCS or VCS MX software:

- Do not specify the -v option for `altera_lnsim.sv` because it defines a systemverilog package.
- Add `-verilog` and `+verilog2001ext+.v` options to make sure all **.v** files are compiled as verilog 2001 files, and all other files are compiled as systemverilog files.
- Add the `-lca` option for Stratix V and later families because they include IEEE-encrypted simulation files for VCS and VCS MX.
- Add `-timescale=1ps/1ps` to ensure picosecond resolution.

3.2.1 Simulating Transport Delays

By default, the VCS and VCS MX software filter out all pulses that are shorter than the propagation delay between primitives. Turning on the **transport delay** options in the VCS and VCS MX software prevents the simulator from filtering out these pulses. Intel Arria 10 devices do not support timing simulation.

Table 11. Transport Delay Simulation Options (VCS and VCS MX)

Option	Description
`+transport_path_delays`	Use when simulation pulses are shorter than the delay in a gate-level primitive. You must include the `+pulse_e/number` and `+pulse_r/number` options.
`+transport_int_delays`	Use when simulation pulses are shorter than the interconnect delay between gate-level primitives. You must include the `+pulse_int_e/number` and `+pulse_int_r/number` options.

Note: The `+transport_path_delays` and `+transport_path_delays` options apply automatically during NativeLink gate-level timing simulation.

The following VCS and VCS MX software command runs a post-synthesis simulation:

```
vcs -R <testbench>.v <gate-level netlist>.v -v <Intel FPGA device family \
library>.v +transport_int_delays +pulse_int_e/0 +pulse_int_r/0 \
+transport_path_delays +pulse_e/0 +pulse_r/0
```

3.2.2 Disabling Timing Violation on Registers

In certain situations, you may want to ignore timing violations on registers and disable the “X” propagation that occurs. For example, this technique may be helpful to eliminate timing violations in internal synchronization registers in asynchronous clock-domain crossing. Intel Arria 10 devices do not support timing simulation. Intel Arria 10 devices do not support timing simulation.

By default, the **x_on_violation_option** logic option is enabled for all design registers, resulting in an output of “X” at timing violation. To disable “X” propagation at timing violations on a specific register, disable the **x_on_violation_option** logic option for the specific register, as shown in the following example from the Intel Quartus Prime Settings File (`.qsf`).

```
set_instance_assignment -name X_ON_VIOLATION_OPTION OFF -to \
<register_name>
```

3.2.3 Generating Power Analysis Files

You can generate a Verilog Value Change Dump File (.vcd) for power analysis in the Intel Quartus Prime software, and then run the **.vcd** from the VCS software. Use this **.vcd** for power analysis in the Intel Quartus Prime power analyzer.

To generate and use a **.vcd** for power analysis, follow these steps:

1. In the Intel Quartus Prime software, click **Assignments ➤ Settings**.
2. Under **EDA Tool Settings**, click **Simulation**.
3. Turn on **Generate Value Change Dump file script**, specify the type of output signals to include, and specify the top-level design instance name in your testbench.
4. Click **Processing ➤ Start Compilation**.
5. Use the following command to include the script in your testbench where the design under test (DUT) is instantiated:
   `include <`*`revision_name`*`>_dump_all_vcd_nodes.v`

   Note: Include the script within the testbench module block. If you include the script outside of the testbench module block, syntax errors occur during compilation.
6. Run the simulation with the VCS command. Exit the VCS software when the simulation is finished and the <*revision_name*>**.vcd** file is generated in the simulation directory.

3.3 VCS Simulation Setup Script Example

The Intel Quartus Prime software can generate a simulation setup script for IP cores in your design. The scripts contain shell commands that compile the required simulation models in the correct order, elaborate the top-level design, and run the simulation for 100 time units by default. You can run these scripts from a Linux command shell.

The scripts for VCS and VCS MX are **vcs_setup.sh** (for Verilog HDL or SystemVerilog) and **vcsmx_setup.sh** (combined Verilog HDL and SystemVerilog with VHDL). Read the generated **.sh** script to see the variables that are available for override when sourcing the script or redefining directly if you edit the script. To set up the simulation for a design, use the command-line to pass variable values to the shell script.

Example 2. Using Command-line to Pass Simulation Variables

```
sh vcsmx_setup.sh\
USER_DEFINED_ELAB_OPTIONS=+rad\
USER_DEFINED_SIM_OPTIONS=+vcs+lic+wait
```

Example 3. Example Top-Level Simulation Shell Script for VCS-MX

```
# Run generated script to compile libraries and IP simulation files
# Skip elaboration and simulation of the IP variation
sh ./ip_top_sim/synopsys/vcsmx/vcsmx_setup.sh SKIP_ELAB=1 SKIP_SIM=1
QSYS_SIMDIR="./ip_top_sim"
#Compile top-level testbench that instantiates IP
vlogan -sverilog ./top_testbench.sv
```

```
#Elaborate and simulate the top-level design
vcs –lca –t ps <elaboration control options> top_testbench
simv <simulation control options>
```

Example 4. Example Top-Level Simulation Shell Script for VCS

```
# Run script to compile libraries and IP simulation files
sh ./ip_top_sim/synopsys/vcs/vcs_setup.sh TOP_LEVEL_NAME=”top_testbench”\
# Pass VCS elaboration options to compile files and elaborate top-level
 passed to the script as the TOP_LEVEL_NAME
USER_DEFINED_ELAB_OPTIONS="top_testbench.sv"\
# Pass in simulation options and run the simulation for specified amount of
time.
USER_DEFINED_SIM_OPTIONS=”<simulation control options>
```

3.4 Document Revision History

Table 12. Document Revision History

Date	Version	Changes
2017.11.06	17.1.0	• Stated no support for Intel Arria 10 timing simulation in Simulating Transport Delays and Disabling Timing Violations on Registers topics. • Added Simulation Library Compiler details and another step to Quick Start Example
2016.05.02	16.0.0	• Noted limitations of NativeLink simulation.
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
2014.06.30	14.0.0	• Replaced MegaWizard Plug-In Manager information with IP Catalog.
November 2012	12.1.0	• Relocated general simulation information to Simulating Altera Designs.
June 2012	12.0.0	• Removed survey link.
November 2011	11.0.1	• Changed to new document template.

Related Links

Documentation Archive
For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

4 Cadence* Incisive Enterprise (IES) Support

You can include your supported EDA simulator in the Intel Quartus Primedesign flow. This chapter provides specific guidelines for simulation of Intel Quartus Prime designs with the Cadence Incisive Enterprise (IES) software.

4.1 Quick Start Example (NC-Verilog)

You can adapt the following RTL simulation example to get started quickly with IES:

1. Click **View ➤ TCL Console** to open the **TCL Console**.
2. To specify your EDA simulator and executable path, type the following Tcl package command in the Intel Quartus Prime tcl shell window:

   `set_user_option -name EDA_TOOL_PATH_NCSIM` <*ncsim executable path*>

   `set_global_assignment -name EDA_SIMULATION_TOOL "NC-Verilog (Verilog)"`

3. Compile simulation model libraries using one of the following methods:
   - Run NativeLink RTL simulation to compile required design files, simulation models, and run your simulator. Verify results in your simulator. If you complete this step you can ignore the remaining steps.
   - To automatically compile all required simulation model libraries for your design in your supported simulator, click **Tools ➤ Launch Simulation Library Compiler**. Specify options for your simulation tool, language, target device family, and output location, and then click **OK**.
   - Map Intel FPGA simulation libraries by adding the following commands to a cds.lib file:

     ```
     include ${CDS_INST_DIR}/tools/inca/files/cds.lib
     DEFINE <lib1>_ver <lib1_ver>
     ```

     Then, compile Intel FPGA simulation models manually:

     ```
      vlog -work <lib1_ver>
     ```

   Use the compiled simulation model libraries during simulatation of your design. Refer to your EDA simulator's documentation for information about running simulation.
4. Elaborate your design and testbench with IES:

   ```
   ncelab <work library>.<top-level entity name>
   ```

5. Run the simulation:

   ```
   ncsim <work library>.<top-level entity name>
   ```

4.2 Cadence Incisive Enterprise (IES) Guidelines

The following guidelines apply to simulation of Intel FPGA designs in the IES software:

- Do not specify the -v option for `altera_lnsim.sv` because it defines a systemverilog package.
- Add `-verilog` and `+verilog2001ext+.v` options to make sure all **.v** files are compiled as verilog 2001 files, and all other files are compiled as systemverilog files.
- Add the `-lca` option for Stratix V and later families because they include IEEE-encrypted simulation files for IES.
- Add `-timescale=1ps/1ps` to ensure picosecond resolution.

4.2.1 Using GUI or Command-Line Interfaces

Intel FPGA supports both the IES GUI and command-line simulator interfaces.

To start the IES GUI, type `nclaunch` at a command prompt.

Table 13. Simulation Executables

Program	Function
`ncvlog` `ncvhdl`	`ncvlog` compiles your Verilog HDL code and performs syntax and static semantics checks. `ncvhdl` compiles your VHDL code and performs syntax and static semantics checks.
`ncelab`	Elaborates the design hierarchy and determines signal connectivity.
`ncsdfc`	Performs back-annotation for simulation with VHDL simulators.
`ncsim`	Runs mixed-language simulation. This program is the simulation kernel that performs event scheduling and executes the simulation code.

4.2.2 Elaborating Your Design

The simulator automatically reads the `.sdo` file during elaboration of the Intel Quartus Prime-generated Verilog HDL or SystemVerilog HDL netlist file. The ncelab command recognizes the embedded system task `$sdf_annotate` and automatically compiles and annotates the `.sdo` file by running `ncsdfc` automatically.

VHDL netlist files do not contain system task calls to locate your `.sdf` file; therefore, you must compile the standard `.sdo` file manually. Locate the `.sdo` file in the same directory where you run elaboration or simulation. Otherwise, the `$sdf_annotate` task cannot reference the `.sdo` file correctly. If you are starting an elaboration or simulation from a different directory, you can either comment out the `$sdf_annotate` and annotate the `.sdo` file with the GUI, or add the full path of the `.sdo` file.

Note: If you use NC-Sim for post-fit VHDL functional simulation of a Stratix V design that includes RAM, an elaboration error might occur if the component declaration parameters are not in the same order as the architecture parameters. Use the `-namemap_mixgen` option with the `ncelab` command to match the component declaration parameter and architecture parameter names.

4.2.3 Back-Annotating Simulation Timing Data (VHDL Only)

You can back annotate timing information in a Standard Delay Output File (.sdo) for VHDL simulators. To back annotate the .sdo timing data at the command line, follow these steps:

1. To compile the **.sdo** with the `ncsdfc` program, type the following command at the command prompt. The ncsdfc program generates an *<output name>***.sdf.X** compiled **.sdo** file

   ```
   ncsdfc <project name>_vhd.sdo –output <output name>
   ```

   Note: If you do not specify an output name, ncsdfc uses *<project name>***.sdo.X**

2. Specify the compiled **.sdf** file for the project by adding the following command to an ASCII SDF command file for the project:

   ```
   COMPILED_SDF_FILE = "<project name>.sdf.X" SCOPE = <instance path>
   ```

3. After compiling the **.sdf** file, type the following command to elaborate the design:

   ```
   ncelab worklib.<project name>:entity –SDF_CMD_FILE <SDF Command File>
   ```

Example 5. Example SDF Command File

```
// SDF command file sdf_file
COMPILED_SDF_FILE = "lpm_ram_dp_test_vhd.sdo.X",
SCOPE = :tb,
MTM_CONTROL = "TYPICAL",
SCALE_FACTORS = "1.0:1.0:1.0",
SCALE_TYPE = "FROM_MTM";
```

4.2.4 Disabling Timing Violation on Registers

In certain situations, you may want to ignore timing violations on registers and disable the "X" propagation that occurs. For example, this technique may be helpful to eliminate timing violations in internal synchronization registers in asynchronous clock-domain crossing. Intel Arria 10 devices do not support timing simulation.

By default, the **x_on_violation_option** logic option is enabled for all design registers, resulting in an output of "X" at timing violation. To disable "X" propagation at timing violations on a specific register, disable the **x_on_violation_option** logic option for the specific register, as shown in the following example from the Intel Quartus Prime Settings File (`.qsf`).

```
set_instance_assignment -name X_ON_VIOLATION_OPTION OFF -to \
<register_name>
```

4.2.5 Simulating Pulse Reject Delays

By default, the IES software filters out all pulses that are shorter than the propagation delay between primitives. Setting the pulse reject delays options in the IES software prevents the simulation tool from filtering out these pulses. Use the following options to ensure that all signal pulses are seen in the simulation results.

Table 14. Pulse Reject Delay Options

Program	Function
`-PULSE_R`	Use when simulation pulses are shorter than the delay in a gate-level primitive. The argument is the percentage of delay for pulse reject limit for the path
`-PULSE_INT_R`	Use when simulation pulses are shorter than the interconnect delay between gate-level primitives. The argument is the percentage of delay for pulse reject limit for the path

4.2.6 Viewing Simulation Waveforms

IES generates a `.trn` file automatically following simulation. You can use the `.trn` for generating the SimVision waveform view.

To view a waveform from a `.trn` file through SimVision, follow these steps:

1. Type `simvision` at the command line. The **Design Browser** dialog box appears.
2. Click **File ➤ Open Database** and click the `.trn` file.
3. In the **Design Browser** dialog box, select the signals that you want to observe from the Hierarchy.
4. Right-click the selected signals and click **Send to Waveform Window**.

   You cannot view a waveform from a `.vcd` file in SimVision, and the `.vcd` file cannot be converted to a `.trn` file.

4.3 IES Simulation Setup Script Example

The Intel Quartus Prime software can generate a `ncsim_setup.sh` simulation setup script for IP cores in your design. The script contains shell commands that compile the required device libraries, IP, or Platform Designer (Standard) simulation models in the correct order. The script then elaborates the top-level design and runs the simulation for 100 time units by default. You can run these scripts from a Linux command shell. To set up the simulation script for a design, you can use the command-line to pass variable values to the shell script.

Read the generated `.sh` script to see the variables that are available for you to override when you source the script or that you can redefine directly in the generated .sh script. For example, you can specify additional elaboration and simulation options with the variables `USER_DEFINED_ELAB_OPTIONS` and `USER_DEFINED_SIM_OPTIONS`.

Example 6. Example Top-Level Simulation Shell Script for Incisive (NCSIM)

```
# Run script to compile libraries and IP simulation files
# Skip elaboration and simulation of the IP variation
sh ./ip_top_sim/cadence/ncsim_setup.sh SKIP_ELAB=1 SKIP_SIM=1 QSYS_SIMDIR="./
ip_top_sim"

#Compile the top-level testbench that instantiates your IP
ncvlog -sv ./top_testbench.sv
#Elaborate and simulate the top-level design
ncelab <elaboration control options> top_testbench
ncsim <simulation control options> top_testbench
```

4.4 Document Revision History

Table 15. Document Revision History

Date	Version	Changes
2017.11.06	17.1.0	• Stated no support for Intel Arria 10 timing simulation in Simulating Transport Delays and Disabling Timing Violations on Registers topics. • Added Simulation Library Compiler details and another step to Quick Start Example
2016.05.02	16.0.0	• Noted limitations of NativeLink simulation.
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
2014.08.18	14.0.a10.0	• Corrected incorrect references to VCS and VCS MX.
2014.06.30	14.0.0	• Replaced MegaWizard Plug-In Manager information with IP Catalog.
November 2012	12.1.0	• Relocated general simulation information to Simulating Altera Designs.
June 2012	12.0.0	• Removed survey link.
November 2011	11.0.1	• Changed to new document template.

Related Links

Documentation Archive
For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

5 Aldec* Active-HDL and Riviera-PRO Support

You can include your supported EDA simulator in the Intel Quartus Prime design flow. This chapter provides specific guidelines for simulation of Intel Quartus Prime designs with the Aldec Active-HDL or Riviera-PRO software.

5.1 Quick Start Example (Active-HDL VHDL)

You can adapt the following RTL simulation example to get started quickly with Active-HDL:

1. To specify your EDA simulator and executable path, type the following Tcl package command in the Intel Quartus Prime tcl shell window:

   `set_user_option -name EDA_TOOL_PATH_ACTIVEHDL` *<Active HDL executable path>*

   `set_global_assignment -name EDA_SIMULATION_TOOL "Active-HDL (VHDL)"`

2. Compile simulation model libraries using one of the following methods:
   - Run NativeLink RTL simulation to compile required design files, simulation models, and run your simulator. Verify results in your simulator. If you complete this step you can ignore the remaining steps.
   - To automatically compile all required simulation model libraries for your design in your supported simulator, click **Tools ➤ Launch Simulation Library Compiler**. Specify options for your simulation tool, language, target device family, and output location, and then click **OK**.
   - Compile Intel FPGA simulation models manually:

     ```
     vlib <library1> <altera_library1>
     vcom -strict93 -dbg -work <library1> <lib1_component/pack.vhd>
     <lib1.vhd>
     ```

   Use the compiled simulation model libraries during simulatation of your design. Refer to your EDA simulator's documentation for information about running simulation.
3. Open the Active-HDL simulator.
4. Create and open the workspace:

   ```
   createdesign <workspace name> <workspace path>
   opendesign -a <workspace name>.adf
   ```

5. Create the work library and compile the netlist and testbench files:

```
vlib work
vcom  -strict93  -dbg -work work <output netlist> <testbench file>
```

6. Load the design:

```
vsim +access+r -t 1ps +transport_int_delays +transport_path_delays \
-L work -L <lib1> -L <lib2> work.<testbench module name>
```

7. Run the simulation in the Active-HDL simulator.

5.2 Aldec Active-HDL and Riviera-PRO Guidelines

The following guidelines apply to simulating Intel FPGA designs in the Active-HDL or Riviera-PRO software.

5.2.1 Compiling SystemVerilog Files

If your design includes multiple SystemVerilog files, you must compile the System Verilog files together with a single alog command. If you have Verilog files and SystemVerilog files in your design, you must first compile the Verilog files, and then compile only the SystemVerilog files in the single `alog` command.

5.2.2 Simulating Transport Delays

By default, the Active-HDL or Riviera-PRO software filters out all pulses that are shorter than the propagation delay between primitives. Turning on the **transport delay** options in the in the Active-HDL or Riviera-PRO software prevents the simulator from filtering out these pulses. Intel Arria 10 devices do not support timing simulation.

Table 16. Transport Delay Simulation Options

Option	Description
`+transport_path_delays`	Use when simulation pulses are shorter than the delay in a gate-level primitive. You must include the `+pulse_e/number` and `+pulse_r/number` options.
`+transport_int_delays`	Use when simulation pulses are shorter than the interconnect delay between gate-level primitives. You must include the `+pulse_int_e/number` and `+pulse_int_r/number` options.

Note: The `+transport_path_delays` and `+transport_path_delays` options apply automatically during NativeLink gate-level timing simulation.

To perform a gate-level timing simulation with the device family library, type the Active-HDL command:

```
vsim -t 1ps -L stratixii -sdftyp /i1=filtref_vhd.sdo \
work.filtref_vhd_vec_tst +transport_int_delays +transport_path_delays
```

5.2.3 Disabling Timing Violation on Registers

In certain situations, you may want to ignore timing violations on registers and disable the “X” propagation that occurs. For example, this technique may be helpful to eliminate timing violations in internal synchronization registers in asynchronous clock-domain crossing. Intel Arria 10 devices do not support timing simulation.

By default, the **x_on_violation_option** logic option is enabled for all design registers, resulting in an output of "X" at timing violation. To disable "X" propagation at timing violations on a specific register, disable the **x_on_violation_option** logic option for the specific register, as shown in the following example from the Intel Quartus Prime Settings File (`.qsf`).

```
set_instance_assignment -name X_ON_VIOLATION_OPTION OFF -to \
<register_name>
```

5.3 Using Simulation Setup Scripts

The Intel Quartus Prime software generates the `rivierapro_setup.tcl` simulation setup script for IP cores in your design. The use and content of the script file is similar to the `msim_setup.tcl` file used by the ModelSim simulator.

Related Links

Simulating IP Cores

5.4 Document Revision History

Table 17. Document Revision History

Date	Version	Changes
2017.11.06	17.1.0	• Stated no support for Intel Arria 10 timing simulation in Simulating Transport Delays and Disabling Timing Violations on Registers topics. • Added Simulation Library Compiler details to Quick Start Example
2016.05.02	16.0.0	• Noted limitations of NativeLink simulation.
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
2014.06.30	14.0.0	• Replaced MegaWizard Plug-In Manager information with IP Catalog.
November 2012	12.1.0	• Relocated general simulation information to Simulating Altera Designs.
June 2012	12.0.0	• Removed survey link.
November 2011	11.0.1	• Changed to new document template.

Related Links

Documentation Archive
For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

6 Design Debugging Using In-System Sources and Probes

The Signal Tap Logic Analyzer and Signal Probe allow you to read or “tap” internal logic signals during run time as a way to debug your logic design.

Traditional debugging techniques often involve using an external pattern generator to exercise the logic and a logic analyzer to study the output waveforms during run time.

You can make the debugging cycle more efficient when you can drive any internal signal manually within your design, which allows you to perform the following actions:

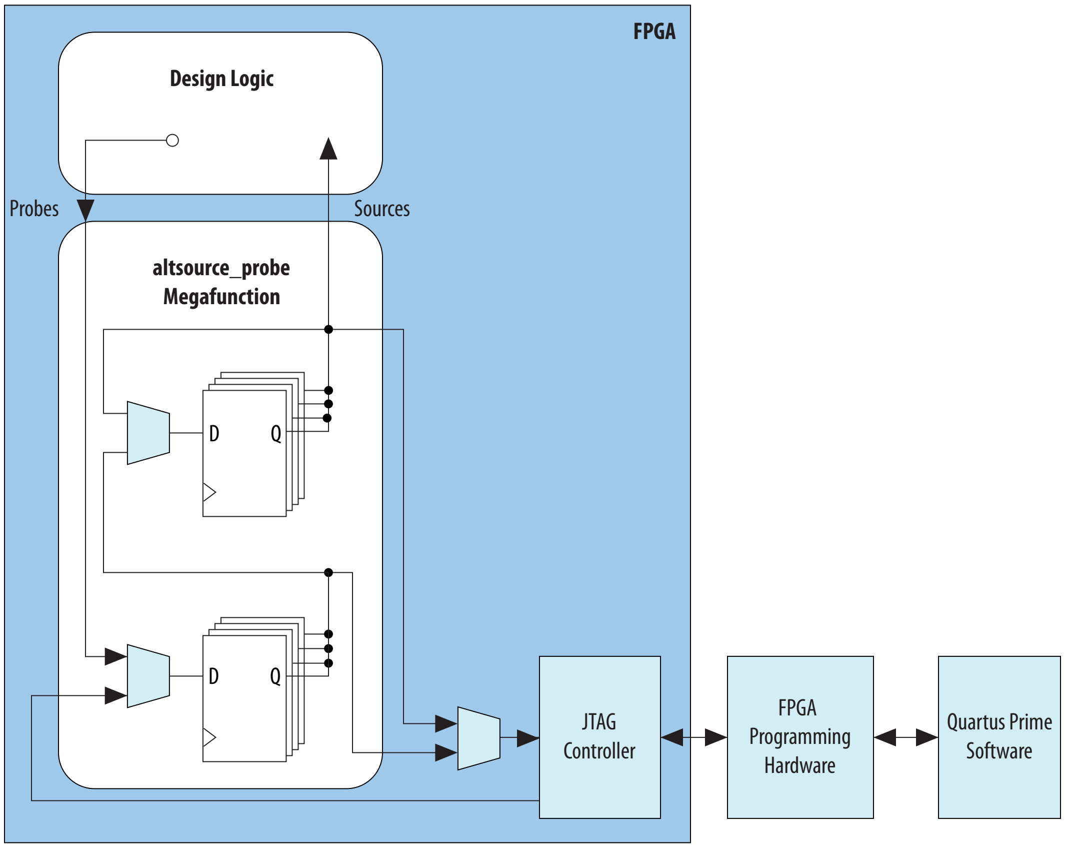
- Force the occurrence of trigger conditions set up in the Signal Tap Logic Analyzer
- Create simple test vectors to exercise your design without using external test equipment
- Dynamically control run time control signals with the JTAG chain

The In-System Sources and Probes Editor in the Intel Quartus Prime software extends the portfolio of verification tools, and allows you to easily control any internal signal and provides you with a completely dynamic debugging environment. Coupled with either the Signal Tap Logic Analyzer or Signal Probe, the In-System Sources and Probes Editor gives you a powerful debugging environment in which to generate stimuli and solicit responses from your logic design.

The Virtual JTAG IP core and the In-System Memory Content Editor also give you the capability to drive virtual inputs into your design. The Intel Quartus Prime software offers a variety of on-chip debugging tools.

The In-System Sources and Probes Editor consists of the ALTSOURCE_PROBE IP core and an interface to control the ALTSOURCE_PROBE IP core instances during run time. Each ALTSOURCE_PROBE IP core instance provides you with source output ports and probe input ports, where source ports drive selected signals and probe ports sample selected signals. When you compile your design, the ALTSOURCE_PROBE IP core sets up a register chain to either drive or sample the selected nodes in your logic design. During run time, the In-System Sources and Probes Editor uses a JTAG connection to shift data to and from the ALTSOURCE_PROBE IP core instances. The figure shows a block diagram of the components that make up the In-System Sources and Probes Editor.

Figure 2. In-System Sources and Probes Editor Block Diagram



The ALTSOURCE_PROBE IP core hides the detailed transactions between the JTAG controller and the registers instrumented in your design to give you a basic building block for stimulating and probing your design. Additionally, the In-System Sources and Probes Editor provides single-cycle samples and single-cycle writes to selected logic nodes. You can use this feature to input simple virtual stimuli and to capture the current value on instrumented nodes. Because the In-System Sources and Probes Editor gives you access to logic nodes in your design, you can toggle the inputs of low-level components during the debugging process. If used in conjunction with the Signal Tap Logic Analyzer, you can force trigger conditions to help isolate your problem and shorten your debugging process.

The In-System Sources and Probes Editor allows you to easily implement control signals in your design as virtual stimuli. This feature can be especially helpful for prototyping your design, such as in the following operations:

- Creating virtual push buttons
- Creating a virtual front panel to interface with your design
- Emulating external sensor data
- Monitoring and changing run time constants on the fly

The In-System Sources and Probes Editor supports Tcl commands that interface with all your ALTSOURCE_PROBE IP core instances to increase the level of automation.

Related Links

- System Debugging Tools
  For an overview and comparison of all the tools available in the Intel Quartus Prime software on-chip debugging tool suite
- System Debugging Tools
  For an overview and comparison of all the tools available in the Intel Quartus Prime software on-chip debugging tool suite

6.1 Hardware and Software Requirements

The following components are required to use the In-System Sources and Probes Editor:

- Intel Quartus Prime software

or

- Intel Quartus Prime Lite Edition
- Download Cable (USB-Blaster^{TM} download cable or ByteBlaster^{TM} cable)
- Intel FPGA development kit or user design board with a JTAG connection to device under test

The In-System Sources and Probes Editor supports the following device families:

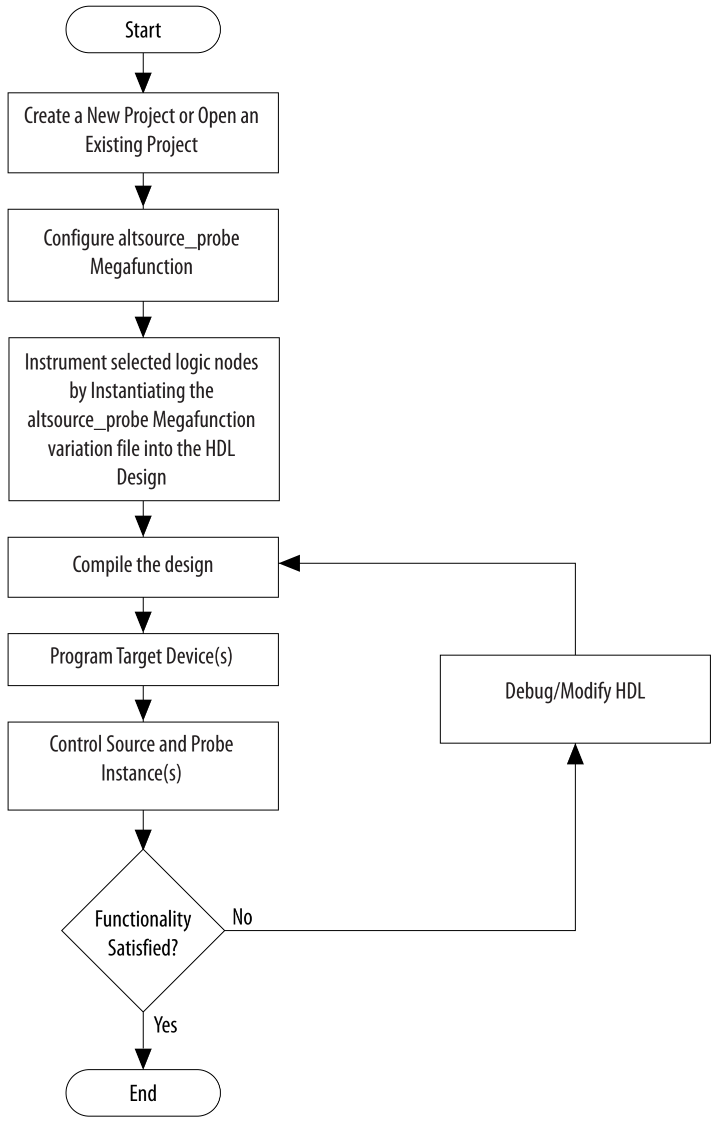
- Arria® series
- Stratix® series
- Cyclone® series
- MAX® series

6.2 Design Flow Using the In-System Sources and Probes Editor

The In-System Sources and Probes Editor supports an RTL flow. Signals that you want to view in the In-System Sources and Probes editor are connected to an instance of the In-System Sources and Probes IP core.

After you compile the design, you can control each instance via the **In-System Sources and Probes Editor** pane or via a Tcl interface.

Figure 3. FPGA Design Flow Using the In-System Sources and Probes Editor


Start
Create a New Project or Open an Existing Project
Configure altsource_probe Megafunction
Instrument selected logic nodes by Instantiating the altsource_probe Megafunction variation file into the HDL Design
Compile the design
Program Target Device(s)
Control Source and Probe Instance(s)
Functionality Satisfied?
No
Debug/Modify HDL
Yes
End

6.2.1 Instantiating the In-System Sources and Probes IP Core

You must instantiate the In-System Sources and Probes IP core before you can use the In-System Sources and Probes editor. Use the IP Catalog and parameter editor to instantiate a custom variation of the In-System Sources and Probes IP core.

To configure the In-System Sources and Probes IP core, perform the following steps::

1. On the Tools menu, click **Tools > IP Catalog**.
2. Locate and double-click the In-System Sources and Probes IP core. The parameter editor appears.
3. Specify a name for your custom IP variation.
4. Specify the desired parameters for your custom IP variation. You can specify up to up to 512 bits for each source. Your design may include up to 128 instances of this IP core.
5. Click **Generate** or **Finish** to generate IP core synthesis and simulation files matching your specifications. The parameter editor generates the necessary variation files and the instantiation template based on your specification. Use the generated template to instantiate the In-System Sources and Probes IP core in your design.

   Note: The In-System Sources and Probes Editor does not support simulation. You must remove the In-System Sources and Probes IP core before you create a simulation netlist.

6.2.2 In-System Sources and Probes IP Core Parameters

Use the template to instantiate the variation file in your design.

Table 18. In-System Sources and Probes IP Port Information

Port Name	Required?	Direction	Comments
`probe[]`	No	Input	The outputs from your design.
`source_clk`	No	Input	Source Data is written synchronously to this clock. This input is required if you turn on **Source Clock** in the **Advanced Options** box in the parameter editor.
`source_ena`	No	Input	Clock enable signal for `source_clk`. This input is required if specified in the **Advanced Options** box in the parameter editor.
`source[]`	No	Output	Used to drive inputs to user design.

You can include up to 128 instances of the in-system sources and probes IP core in your design, if your device has available resources. Each instance of the IP core uses a pair of registers per signal for the width of the widest port in the IP core. Additionally, there is some fixed overhead logic to accommodate communication between the IP core instances and the JTAG controller. You can also specify an additional pair of registers per source port for synchronization.

You can use the Intel Quartus Prime incremental compilation feature to reduce compilation time. Incremental compilation allows you to organize your design into logical partitions. During recompilation of a design, incremental compilation preserves the compilation results and performance of unchanged partitions and reduces design iteration time by compiling only modified design partitions.

6.3 Compiling the Design

When you compile your design that includes the In-System Sources and ProbesIP core, the In-System Sources and Probes and SLD Hub Controller IP core are added to your compilation hierarchy automatically. These IP cores provide communication between the JTAG controller and your instrumented logic.

You can modify the number of connections to your design by editing the In-System Sources and Probes IP core. To open the design instance you want to modify in the parameter editor, double-click the instance in the Project Navigator. You can then modify the connections in the HDL source file. You must recompile your design after you make changes.

You can use the Intel Quartus Prime incremental compilation feature to reduce compilation design into logical partitions. During recompilation of a design, incremental compilation preserves the compilation results and performance of unchanged partitions and reduces design iteration time by compiling only modified design partitions.

6.4 Running the In-System Sources and Probes Editor

The In-System Sources and Probes Editor gives you control over all ALTSOURCE_PROBE IP core instances within your design. The editor allows you to view all available run time controllable instances of the ALTSOURCE_PROBE IP core in your design, provides a push-button interface to drive all your source nodes, and provides a logging feature to store your probe and source data.

To run the In-System Sources and Probes Editor:

- On the **Tools** menu, click **In-System Sources and Probes Editor.**

6.4.1 In-System Sources and Probes Editor GUI

The In-System Sources and Probes Editor contains three panes:

- **JTAG Chain Configuration**—Allows you to specify programming hardware, device, and file settings that the In-System Sources and Probes Editor uses to program and acquire data from a device.
- **Instance Manager**—Displays information about the instances generated when you compile a design, and allows you to control data that the In-System Sources and Probes Editor acquires.
- **In-System Sources and Probes Editor**—Logs all data read from the selected instance and allows you to modify source data that is written to your device.

When you use the In-System Sources and Probes Editor, you do not need to open a Intel Quartus Prime software project. The In-System Sources and Probes Editor retrieves all instances of the ALTSOURCE_PROBE IP core by scanning the JTAG chain and sending a query to the device selected in the **JTAG Chain Configuration** pane. You can also use a previously saved configuration to run the In-System Sources and Probes Editor.

Each **In-System Sources and Probes Editor** pane can access the ALTSOURCE_PROBE IP core instances in a single device. If you have more than one device containing IP core instances in a JTAG chain, you can launch multiple **In-System Sources and Probes Editor** panes to access the IP core instances in each device.

6.4.2 Programming Your Device With JTAG Chain Configuration

After you compile your project, you must configure your FPGA before you use the In-System Sources and Probes Editor.

To configure a device to use with the In-System Sources and Probes Editor, perform the following steps:

1. Open the In-System Sources and Probes Editor.
2. In the **JTAG Chain Configuration** pane, point to **Hardware,** and then select the hardware communications device. You may be prompted to configure your hardware; in this case, click **Setup**.
3. From the **Device** list, select the FPGA device to which you want to download the design (the device may be automatically detected). You may need to click **Scan Chain** to detect your target device.
4. In the **JTAG Chain Configuration** pane, click to browse for the SRAM Object File (**.sof**) that includes the In-System Sources and Probes instance or instances. (The **.sof** may be automatically detected).
5. Click **Program Device** to program the target device.

6.4.3 Instance Manager

The **Instance Manager** pane provides a list of all ALTSOURCE_PROBE instances in the design and allows you to configure how data is acquired from or written to those instances.

The following buttons and sub-panes are provided in the **Instance Manager** pane:

- **Read Probe Data**—Samples the probe data in the selected instance and displays the probe data in the **In-System Sources and Probes Editor** pane.
- **Continuously Read Probe Data**—Continuously samples the probe data of the selected instance and displays the probe data in the **In-System Sources and Probes Editor** pane; you can modify the sample rate via the **Probe read interval** setting.
- **Stop Continuously Reading Probe Data**—Cancels continuous sampling of the probe of the selected instance.
- **Read Source Data**—Reads the data of the sources in the selected instances.
- **Probe Read Interval**—Displays the sample interval of all the In-System Sources and Probe instances in your design; you can modify the sample interval by clicking **Manual.**
- **Event Log**—Controls the event log in the **In-System Sources and Probes Editor** pane.
- **Write Source Data**—Allows you to manually or continuously write data to the system.

The status of each instance is also displayed beside each entry in the **Instance Manager** pane. The status indicates if the instance is **Not running Offloading data**, **Updating data**, or if an **Unexpected JTAG communication error** occurs. This status indicator provides information about the sources and probes instances in your design.

6.4.4 In-System Sources and Probes Editor Pane

The **In-System Sources and Probes Editor** pane allows you to view data from all sources and probes in your design.

The data is organized according to the index number of the instance. The editor provides an easy way to manage your signals, and allows you to rename signals or group them into buses. All data collected from in-system source and probe nodes is recorded in the event log and you can view the data as a timing diagram.

6.4.4.1 Reading Probe Data

You can read data by selecting the ALTSOURCE_PROBE instance in the **Instance Manager** pane and clicking **Read Probe Data**.

This action produces a single sample of the probe data and updates the data column of the selected index in the **In-System Sources and Probes Editor** pane. You can save the data to an event log by turning on the **Save data to event log** option in the **Instance Manager** pane.

If you want to sample data from your probe instance continuously, in the **Instance Manager** pane, click the instance you want to read, and then click **Continuously read probe data**. While reading, the status of the active instance shows **Unloading**. You can read continuously from multiple instances.

You can access read data with the shortcut menus in the **Instance Manager** pane.

To adjust the probe read interval, in the **Instance Manager** pane, turn on the **Manual** option in the **Probe read interval** sub-pane, and specify the sample rate in the text field next to the **Manual** option. The maximum sample rate depends on your computer setup. The actual sample rate is shown in the **Current interval** box. You can adjust the event log window buffer size in the **Maximum Size** box.

6.4.4.2 Writing Data

To modify the source data you want to write into the ALTSOURCE_PROBE instance, click the name field of the signal you want to change. For buses of signals, you can double-click the data field and type the value you want to drive out to the ALTSOURCE_PROBE instance. The In-System Sources and Probes Editor stores the modified source data values in a temporary buffer.

Modified values that are not written out to the ALTSOURCE_PROBE instances appear in red. To update the ALTSOURCE_PROBE instance, highlight the instance in the **Instance Manager** pane and click **Write source data**. The **Write source data** function is also available via the shortcut menus in the **Instance Manager** pane.

The In-System Sources and Probes Editor provides the option to continuously update each ALTSOURCE_PROBE instance. Continuous updating allows any modifications you make to the source data buffer to also write immediately to the ALTSOURCE_PROBE instances. To continuously update the ALTSOURCE_PROBE instances, change the **Write source data** field from **Manually** to **Continuously**.

6.4.4.3 Organizing Data

The **In-System Sources and Probes Editor** pane allows you to group signals into buses, and also allows you to modify the display options of the data buffer.

To create a group of signals, select the node names you want to group, right-click and select **Group**. You can modify the display format in the Bus Display Format and the Bus Bit order shortcut menus.

The **In-System Sources and Probes Editor** pane allows you to rename any signal. To rename a signal, double-click the name of the signal and type the new name.

The event log contains a record of the most recent samples. The buffer size is adjustable up to 128k samples. The time stamp for each sample is logged and is displayed above the event log of the active instance as you move your pointer over the data samples.

You can save the changes that you make and the recorded data to a Sources and Probes File (**`.spf`**). To save changes, on the File menu, click **Save**. The file contains all the modifications you made to the signal groups, as well as the current data event log.

6.5 Tcl interface for the In-System Sources and Probes Editor

To support automation, the In-System Sources and Probes Editor supports the procedures described in this chapter in the form of Tcl commands. The Tcl package for the In-System Sources and Probes Editor is included by default when you run **quartus_stp**.

The Tcl interface for the In-System Sources and Probes Editor provides a powerful platform to help you debug your design. The Tcl interface is especially helpful for debugging designs that require toggling multiple sets of control inputs. You can combine multiple commands with a Tcl script to define a custom command set.

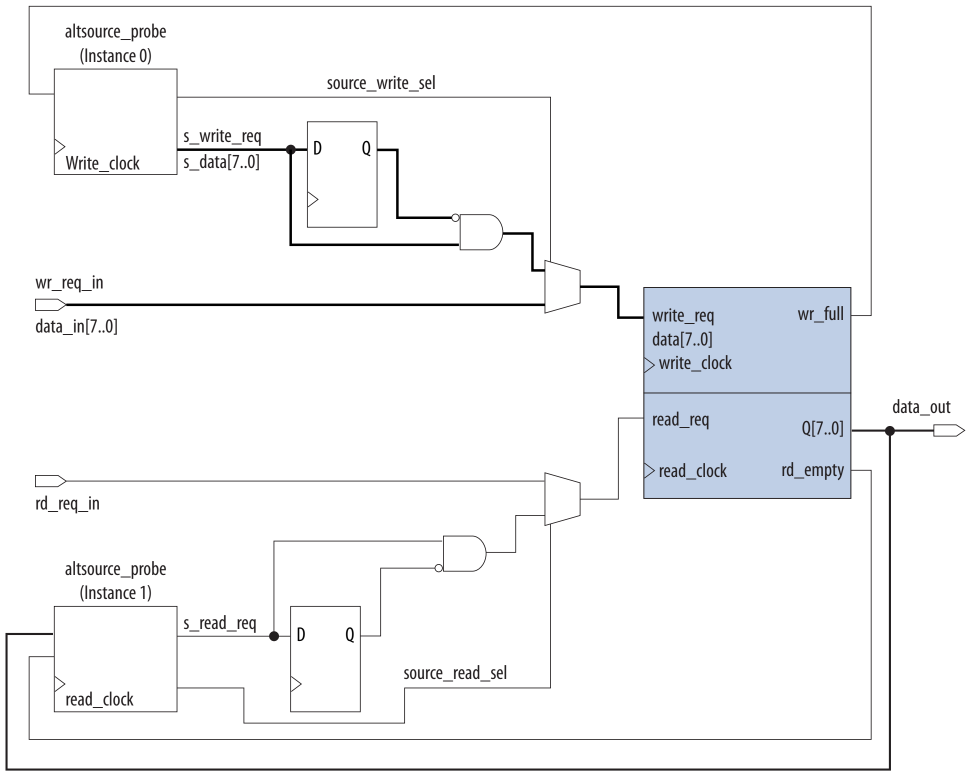
Table 19. In-System Sources and Probes Tcl Commands

Command	Argument	Description
`start_insystem_source_probe`	`-device_name` *<device name>* `-hardware_name` *<hardware name>*	Opens a handle to a device with the specified hardware. Call this command before starting any transactions.
`get_insystem_source_probe_instance_info`	`-device_name` *<device name>* -hardware_name *<hardware name>*	Returns a list of all ALTSOURCE_PROBE instances in your design. Each record returned is in the following format: {*<instance Index>*, *<source width>*, *<probe width>*, *<instance name>*}
`read_probe_data`	`-instance_index` *<instance_index>* `-value_in_hex` (optional)	Retrieves the current value of the probe. A string is returned that specifies the status of each probe, with the MSB as the left-most bit.
`read_source_data`	`-instance_index` *<instance_index>* `-value_in_hex` (optional)	Retrieves the current value of the sources. A string is returned that specifies the status of each source, with the MSB as the left-most bit.
`write_source_data`	`-instance_index` *<instance_index>* `-value` *<value>* `-value_in_hex` (optional)	Sets the value of the sources. A binary string is sent to the source ports, with the MSB as the left-most bit.
`end_insystem_source_probe`	None	Releases the JTAG chain. Issue this command when all transactions are finished.

The example shows an excerpt from a Tcl script with procedures that control the ALTSOURCE_PROBE instances of the design as shown in the figure below. The example design contains a DCFIFO with ALTSOURCE_PROBE instances to read from and write to the DCFIFO. A set of control muxes are added to the design to control the flow of data to the DCFIFO between the input pins and the ALTSOURCE_PROBE instances. A pulse generator is added to the read request and write request control lines to guarantee a single sample read or write. The ALTSOURCE_PROBE instances, when used with the script in the example below, provide visibility into the contents of the FIFO by performing single sample write and read operations and reporting the state of the full and empty status flags.

Use the Tcl script in debugging situations to either empty or preload the FIFO in your design. For example, you can use this feature to preload the FIFO to match a trigger condition you have set up within the Signal Tap Logic Analyzer.

Figure 4. **DCFIFO Example Design Controlled by Tcl Script**



```
## Setup USB hardware  - assumes only USB Blaster is installed and
## an FPGA is the only device in the JTAG chain
set usb [lindex [get_hardware_names] 0]
set device_name [lindex [get_device_names -hardware_name $usb] 0]
## write procedure :  argument value is integer
proc write {value} {
global device_name usb
variable full
start_insystem_source_probe -device_name $device_name -hardware_name $usb
#read full flag
set full [read_probe_data -instance_index 0]
if {$full == 1} {end_insystem_source_probe
return "Write Buffer Full"
}
##toggle select line, drive value onto port, toggle enable
##bits 7:0 of instance 0 is S_data[7:0]; bit 8 = S_write_req;
##bit 9 = Source_write_sel
##int2bits is custom procedure that returns a bitstring from an integer
    ## argument
write_source_data -instance_index 0 -value /[int2bits [expr 0x200 | $value]]
write_source_data -instance_index 0 -value [int2bits [expr 0x300 | $value]]
##clear transaction
write_source_data -instance_index 0 -value 0
end_insystem_source_probe
}
proc read {} {
global device_name usb
variable empty
start_insystem_source_probe -device_name $device_name -hardware_name $usb
##read empty flag : probe port[7:0] reads FIFO output; bit 8 reads empty_flag
set empty [read_probe_data -instance_index 1]
if {[regexp {1........} $empty]} { end_insystem_source_probe
return "FIFO empty" }
## toggle select line for read transaction
## Source_read_sel = bit 0; s_read_reg = bit 1
## pulse read enable on DC FIFO
write_source_data -instance_index 1 -value 0x1 -value_in_hex
write_source_data -instance_index 1 -value 0x3 -value_in_hex
set x [read_probe_data -instance_index 1 ]
end_insystem_source_probe
return $x
}
```

Related Links

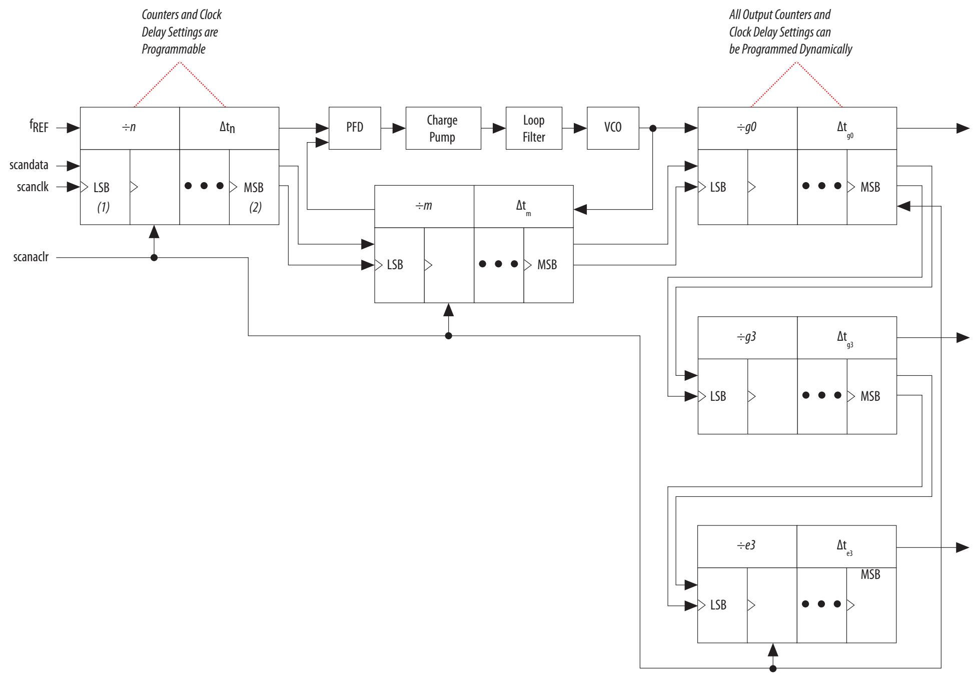
- Tcl Scripting
- Intel Quartus Prime Settings File Manual
- Command Line Scripting
- Tcl Scripting
- Intel Quartus Prime Settings File Manual
- Command Line Scripting

6.6 Design Example: Dynamic PLL Reconfiguration

The In-System Sources and Probes Editor can help you create a virtual front panel during the prototyping phase of your design. You can create relatively simple, high functioning designs of in a short amount of time. The following PLL reconfiguration example demonstrates how to use the In-System Sources and Probes Editor to provide a GUI to dynamically reconfigure a Stratix PLL.

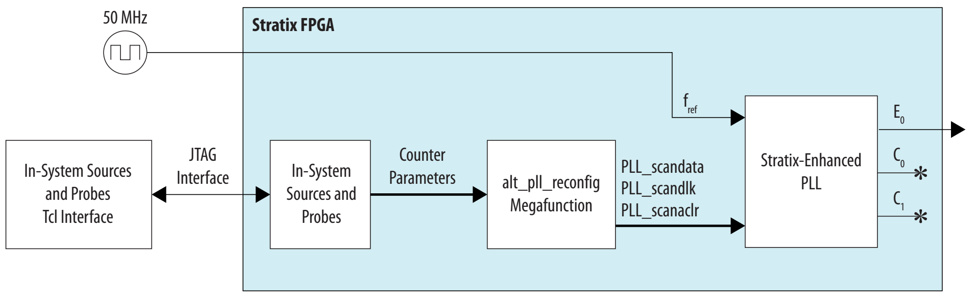
Stratix PLLs allow you to dynamically update PLL coefficients during run time. Each enhanced PLL within the Stratix device contains a register chain that allows you to modify the pre-scale counters (m and n values), output divide counters, and delay counters. In addition, the ALTPLL_RECONFIG IP core provides an easy interface to access the register chain counters. The ALTPLL_RECONFIG IP core provides a cache that contains all modifiable PLL parameters. After you update all the PLL parameters in the cache, the ALTPLL_RECONFIG IP core drives the PLL register chain to update the PLL with the updated parameters. The figure shows a Stratix-enhanced PLL with reconfigurable coefficients.

Figure 5. Stratix-Enhanced PLL with Reconfigurable Coefficients


Counters and Clock Delay Settings are Programmable
All Output Counters and Clock Delay Settings can be Programmed Dynamically
fREF
scandata
scanclk
scanclr
÷n
Δtn
LSB (1)
MSB (2)
PFD
Charge Pump
Loop Filter
VCO
÷m
Δtm
÷g0
Δtg0
÷g3
Δtg3
÷e3
Δte3
LSB
MSB

The following design example uses an ALTSOURCE_PROBE instance to update the PLL parameters in the ALTPLL_RECONFIG IP core cache. The ALTPLL_RECONFIG IP core connects to an enhanced PLL in a Stratix FPGA to drive the register chain containing the PLL reconfigurable coefficients. This design example uses a Tcl/Tk script to generate a GUI where you can enter in new m and n values for the enhanced PLL. The Tcl script extracts the m and n values from the GUI, shifts the values out to the ALTSOURCE_PROBE instances to update the values in the ALTPLL_RECONFIG IP core cache, and asserts the reconfiguration signal on the ALTPLL_RECONFIG IP core. The reconfiguration signal on the ALTPLL_RECONFIG IP core starts the register chain transaction to update all PLL reconfigurable coefficients.

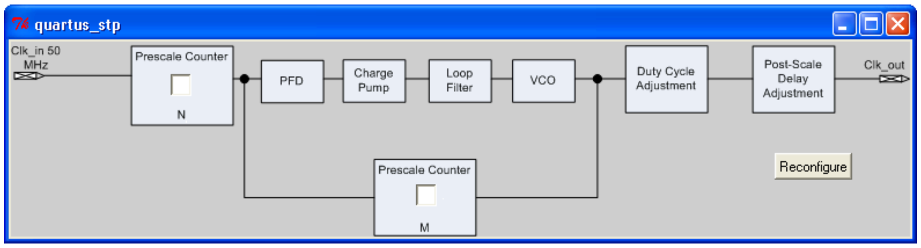
Figure 6. Block Diagram of Dynamic PLL Reconfiguration Design Example



This design example was created using a Nios II Development Kit, Stratix Edition. The file `sourceprobe_DE_dynamic_pll.zip` contains all the necessary files for running this design example, including the following:

- `Readme.txt`—A text file that describes the files contained in the design example and provides instructions about running the Tk GUI shown in the figure below.
- `Interactive_Reconfig.qar`—The archived Intel Quartus Prime project for this design example.

Figure 7. Interactive PLL Reconfiguration GUI Created with Tk and In-System Sources and Probes Tcl Package



Related Links

- On-chip Debugging Design Examples
  to download the In-System Sources and Probes Example
- On-chip Debugging Design Examples
  to download the In-System Sources and Probes Example

6.7 Document Revision History

Table 20. Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
June 2014	14.0.0	Updated formatting.
June 2012	12.0.0	Removed survey link.
November 2011	10.1.1	Template update.
December 2010	10.1.0	Minor corrections. Changed to new document template.
July 2010	10.0.0	Minor corrections.
November 2009	9.1.0	• Removed references to obsolete devices. • Style changes.
March 2009	9.0.0	No change to content.
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	8.0.0	• Documented that this feature does not support simulation on page 17–5 • Updated Figure 17–8 for Interactive PLL reconfiguration manager • Added hyperlinks to referenced documents throughout the chapter • Minor editorial updates

Related Links

Documentation Archive

For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

7 Timing Analysis Overview

7.1 Timing Analysis Overview

Comprehensive static timing analysis involves analysis of register-to-register, I/O, and asynchronous reset paths. Timing analysis with the Timing Analyzer uses data required times, data arrival times, and clock arrival times to verify circuit performance and detect possible timing violations.

The Timing Analyzer determines the timing relationships that must be met for the design to correctly function, and checks arrival times against required times to verify timing. This chapter is an overview of the concepts you need to know to analyze your designs with the Timing Analyzer.

Related Links

The Intel Quartus PrimeTiming Analyzer on page 79
For more information about the Timing Analyzer flow and Timing Analyzer examples.

7.2 Timing Analyzer Terminology and Concepts

Table 21. Timing Analyzer Terminology

Term	Definition
nodes	Most basic timing netlist unit. Used to represent ports, pins, and registers.
cells	Look-up tables (LUT), registers, digital signal processing (DSP) blocks, memory blocks, input/output elements, and so on. *Note:* For Intel Stratix devices, the LUTs and registers are contained in logic elements (LE) and modeled as cells.
pins	Inputs or outputs of cells.
nets	Connections between pins.
ports	Top-level module inputs or outputs; for example, device pins.
clocks	Abstract objects representing clock domains inside or outside of your design.

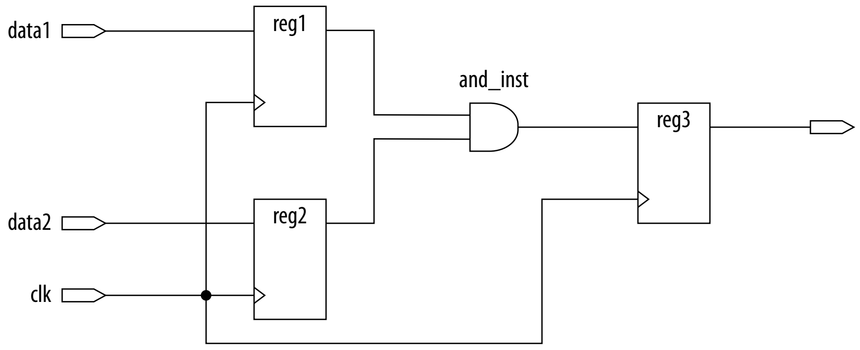
7.2.1 Timing Netlists and Timing Paths

The Timing Analyzer requires a timing netlist to perform timing analysis on any design. After you generate a timing netlist, the Timing Analyzer uses the data to help determine the different design elements in your design and how to analyze timing.

7.2.1.1 The Timing Netlist

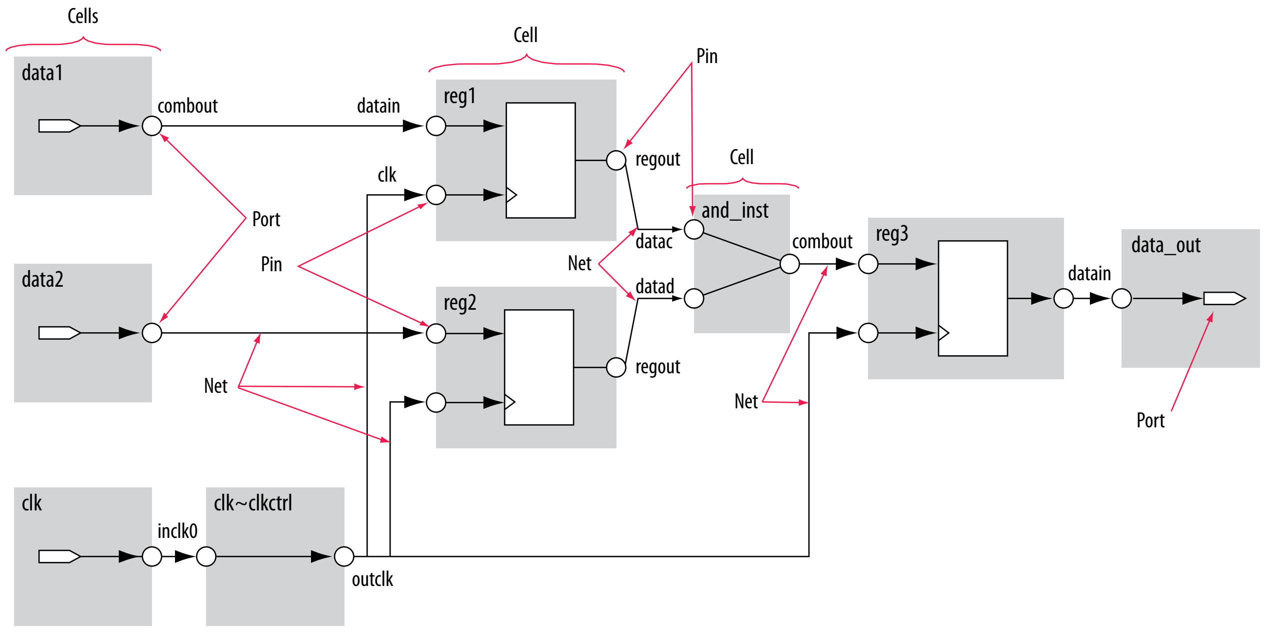
A sample design for which the Timing Analyzer generates a timing netlist equivalent.

Figure 8. Sample Design



The timing netlist for the sample design shows how different design elements are divided into cells, pins, nets, and ports.

Figure 9. The Timing Analyzer Netlist



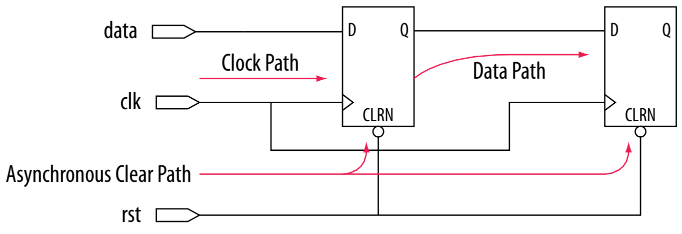
7.2.1.2 Timing Paths

Timing paths connect two design nodes, such as the output of a register to the input of another register.

Understanding the types of timing paths is important to timing closure and optimization. The Timing Analyzer uses the following commonly analyzed paths:

- **Edge paths**—connections from ports-to-pins, from pins-to-pins, and from pins-to-ports.
- **Clock paths**—connections from device ports or internally generated clock pins to the clock pin of a register.
- **Data paths**—connections from a port or the data output pin of a sequential element to a port or the data input pin of another sequential element.
- **Asynchronous paths**—connections from a port or asynchronous pins of another sequential element such as an asynchronous reset or asynchronous clear.

Figure 10. Path Types Commonly Analyzed by the Timing Analyzer



In addition to identifying various paths in a design, the Timing Analyzer analyzes clock characteristics to compute the worst-case requirement between any two registers in a single register-to-register path. You must constrain all clocks in your design before analyzing clock characteristics.

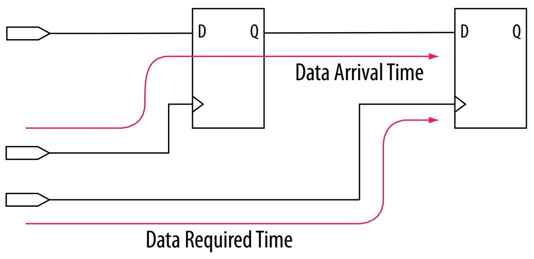
7.2.1.3 Data and Clock Arrival Times

After the Timing Analyzer identifies the path type, it can report data and clock arrival times at register pins.

The Timing Analyzer calculates data arrival time by adding the launch edge time to the delay from the clock source to the clock pin of the source register, the micro clock-to-output delay (μt_{CO}) of the source register, and the delay from the source register's data output (Q) to the destination register's data input (D).

The Timing Analyzer calculates data required time by adding the latch edge time to the sum of all delays between the clock port and the clock pin of the destination register, including any clock port buffer delays, and subtracts the micro setup time (μt_{SU}) of the destination register, where the μt_{SU} is the intrinsic setup time of an internal register in the FPGA.

Figure 11. Data Arrival and Data Required Times



The basic calculations for data arrival and data required times including the launch and latch edges.

Figure 12. Data Arrival and Data Required Time Equations

$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Source Clock Delay} + \mu t_{CO} + \text{Register-to-Register Delay}$$

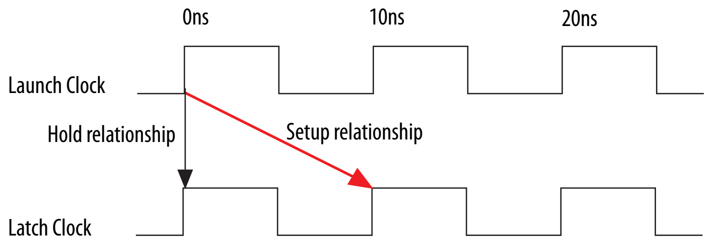
$$\text{Data Required Time} = \text{Latch Edge} + \text{Destination Clock Delay} - \mu t_{SU}$$

7.2.1.4 Launch and Latch Edges

All timing relies on one or more clocks. In addition to analyzing paths, the Timing Analyzer determines clock relationships for all register-to-register transfers in your design.

The following figure shows the launch edge, which is the clock edge that sends data out of a register or other sequential element, and acts as a source for the data transfer. A latch edge is the active clock edge that captures data at the data port of a register or other sequential element, acting as a destination for the data transfer. In this example, the launch edge sends the data from register `reg1` at 0 ns, and the register `reg2` captures the data when triggered by the latch edge at 10 ns. The data arrives at the destination register before the next latch edge.

Figure 13. Setup and Hold Relationship for Launch and Latch Edges 10ns Apart



In timing analysis, and with the Timing Analyzer specifically, you create clock constraints and assign those constraints to nodes in your design. These clock constraints provide the structure required for repeatable data relationships. The primary relationships between clocks, in the same or different domains, are the setup relationship and the hold relationship.

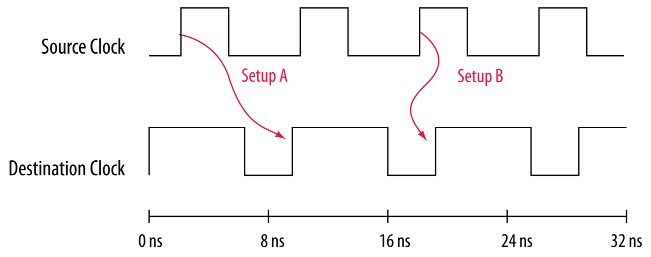
Note: If you do not constrain the clocks in your design, the Intel Quartus Prime software analyzes in terms of a 1 GHz clock to maximize timing based Fitter effort. To ensure realistic slack values, you must constrain all clocks in your design with real values.

7.2.2 Clock Setup Check

To perform a clock setup check, the Timing Analyzer determines a setup relationship by analyzing each launch and latch edge for each register-to-register path.

For each latch edge at the destination register, the Timing Analyzer uses the closest previous clock edge at the source register as the launch edge. The following figure shows two setup relationships, setup A and setup B. For the latch edge at 10 ns, the closest clock that acts as a launch edge is at 3 ns and is labeled setup A. For the latch edge at 20 ns, the closest clock that acts as a launch edge is 19 ns and is labeled setup B. TimQuest analyzes the most restrictive setup relationship, in this case setup B; if that relationship meets the design requirement, then setup A meets it by default.

Figure 14. Setup Check



The Timing Analyzer reports the result of clock setup checks as slack values. Slack is the margin by which a timing requirement is met or not met. Positive slack indicates the margin by which a requirement is met; negative slack indicates the margin by which a requirement is not met.

Figure 15. Clock Setup Slack for Internal Register-to-Register Paths

Clock Setup Slack = Data Required Time – Data Arrival Time

Data Arrival Time = Launch Edge + Clock Network Delay to Source Register + μt_{CO} + Register-to-Register Delay

Data Required Time = Latch Edge + Clock Network Delay to Destination Register – μt_{SU} – Setup Uncertainty

The Timing Analyzer performs setup checks using the maximum delay when calculating data arrival time, and minimum delay when calculating data required time.

Figure 16. Clock Setup Slack from Input Port to Internal Register

Clock Setup Slack = Data Required Time – Data Arrival Time

Data Arrival Time = Launch Edge + Clock Network Delay + Input Maximum Delay + Port-to-Register Delay

Data Required Time = Latch Edge + Clock Network Delay to Destination Register – μt_{SU} – Setup Uncertainty

Figure 17. Clock Setup Slack from Internal Register to Output Port

Clock Setup Slack = Data Required Time – Data Arrival Time

Data Required Time = Latch Edge + Clock Network Delay to Output Port – Output Maximum Delay

Data Arrival Time = Launch Edge + Clock Network Delay to Source Register +μt_{CO} + Register-to-Port Delay

7.2.3 Clock Hold Check

To perform a clock hold check, the Timing Analyzer determines a hold relationship for each possible setup relationship that exists for all source and destination register pairs. The Timing Analyzer checks all adjacent clock edges from all setup relationships to determine the hold relationships.

The Timing Analyzer performs two hold checks for each setup relationship. The first hold check determines that the data launched by the current launch edge is not captured by the previous latch edge. The second hold check determines that the data launched by the next launch edge is not captured by the current latch edge. From the possible hold relationships, the Timing Analyzer selects the hold relationship that is the most restrictive. The most restrictive hold relationship is the hold relationship with the smallest difference between the latch and launch edges and determines the minimum allowable delay for the register-to-register path. In the following example, the Timing Analyzer selects hold check A2 as the most restrictive hold relationship of two setup relationships, setup A and setup B, and their respective hold checks.

Figure 18. Setup and Hold Check Relationships

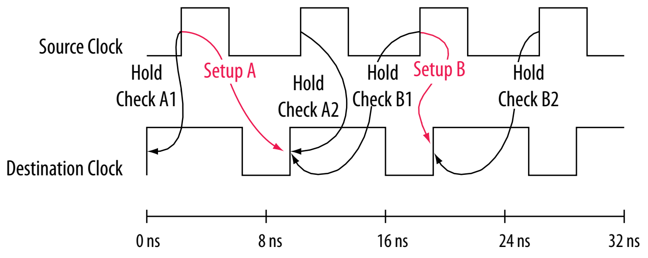


Figure 19. Clock Hold Slack for Internal Register-to-Register Paths

$$\text{Clock Hold Slack} = \text{Data Arrival Time} - \text{Data Required Time}$$
$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay to Source Register} + \mu t_{CO} + \text{Register-to-Register Delay}$$
$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register} + \mu t_{H} + \text{Hold Uncertainty}$$

The Timing Analyzer performs hold checks using the minimum delay when calculating data arrival time, and maximum delay when calculating data required time.

Figure 20. Clock Hold Slack Calculation from Input Port to Internal Register

$$\text{Clock Hold Slack} = \text{Data Arrival Time} - \text{Data Required Time}$$
$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay} + \text{Input Minimum Delay} + \text{Pin-to-Register Delay}$$
$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register} + \mu t_{H}$$

Figure 21. Clock Hold Slack Calculation from Internal Register to Output Port

$$\text{Clock Hold Slack} = \text{Data Arrival Time} - \text{Data Required Time}$$
$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay to Source Register} + \mu t_{CO} + \text{Register-to-Pin Delay}$$
$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay} - \text{Output Minimum Delay}$$

7.2.4 Recovery and Removal Time

Recovery time is the minimum length of time for the deassertion of an asynchronous control signal relative to the next clock edge.

For example, signals such as `clear` and `preset` must be stable before the next active clock edge. The recovery slack calculation is similar to the clock setup slack calculation, but it applies to asynchronous control signals.

Figure 22. Recovery Slack Calculation if the Asynchronous Control Signal is Registered

$$\text{Recovery Slack Time} = \text{Data Required Time} - \text{Data Arrival Time}$$
$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register} - \mu t_{SU}$$
$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay to Source Register} + \mu t_{CO} + \text{Register-to-Register Delay}$$

Figure 23. Recovery Slack Calculation if the Asynchronous Control Signal is not Registered

$$\text{Recovery Slack Time} = \text{Data Required Time} - \text{Data Arrival Time}$$
$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register} - \mu t_{SU}$$
$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay} + \text{Input Maximum Delay} + \text{Port-to-Register Delay}$$

Note: If the asynchronous reset signal is from a device I/O port, you must create an input delay constraint for the asynchronous reset port for the Timing Analyzer to perform recovery analysis on the path.

Removal time is the minimum length of time the deassertion of an asynchronous control signal must be stable after the active clock edge. The Timing Analyzer removal slack calculation is similar to the clock hold slack calculation, but it applies asynchronous control signals.

Figure 24. Removal Slack Calcuation if the Asynchronous Control Signal is Registered

Removal Slack Time	= Data Arrival Time – Data Required Time
Data Arrival Time	= Launch Edge + Clock Network Delay to Source Register + μt_{CO} of Source Register + Register-to-Register Delay
Data Required Time	= Latch Edge + Clock Network Delay to Destination Register + μt_H

Figure 25. Removal Slack Calculation if the Asynchronous Control Signal is not Registered

Removal Slack Time	= Data Arrival Time – Data Required Time
Data Arrival Time	= Launch Edge + Clock Network Delay + Input Minimum Delay of Pin + Minimum Pin-to-Register Delay
Data Required Time	= Latch Edge + Clock Network Delay to Destination Register + μt_H

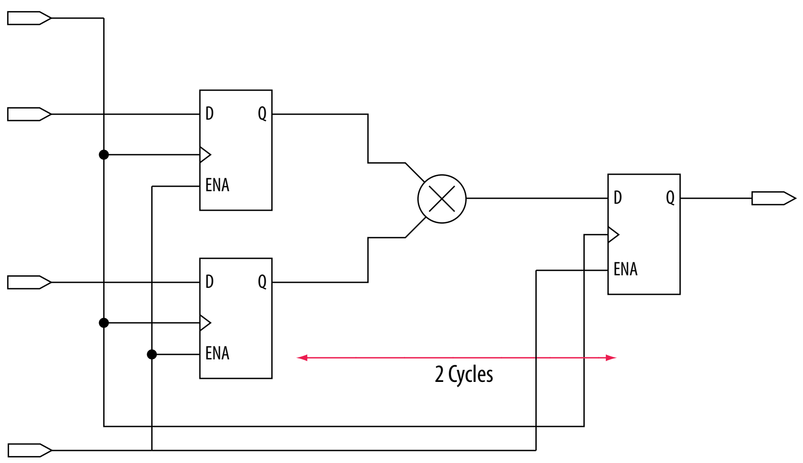
If the asynchronous reset signal is from a device pin, you must assign the **Input Minimum Delay** timing assignment to the asynchronous reset pin for the Timing Analyzer to perform removal analysis on the path.

7.2.5 Multicycle Paths

Multicycle paths are data paths that require a non-default setup and/or hold relationship for proper analysis.

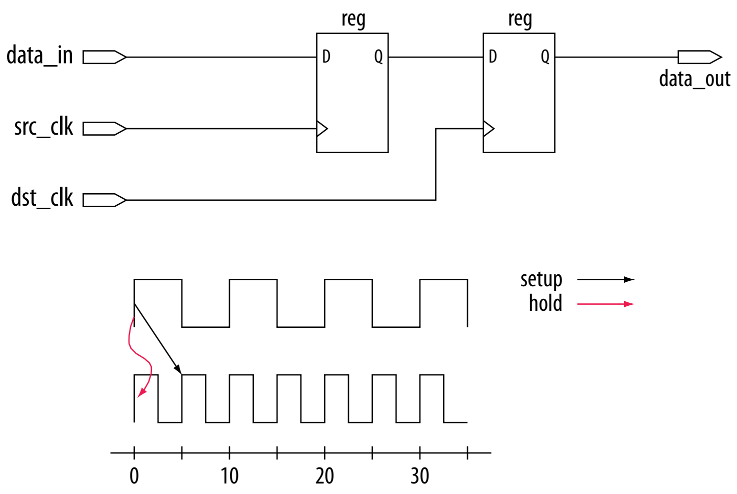
For example, a register may be required to capture data on every second or third rising clock edge. An example of a multicycle path between the input registers of a multiplier and an output register where the destination latches data on every other clock edge.

Figure 26. Multicycle Path



A register-to-register path used for the default setup and hold relationship, the respective timing diagrams for the source and destination clocks, and the default setup and hold relationships, when the source clock, `src_clk`, has a period of 10 ns and the destination clock, `dst_clk`, has a period of 5 ns. The default setup relationship is 5 ns; the default hold relationship is 0 ns.

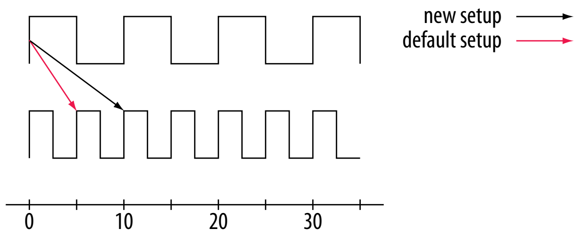
Figure 27. Register-to-Register Path and Default Setup and Hold Timing Diagram



To accommodate the system requirements you can modify the default setup and hold relationships with a multicycle timing exception.

The actual setup relationship after you apply a multicycle timing exception. The exception has a multicycle setup assignment of two to use the second occurring latch edge; in this example, to 10 ns from the default value of 5 ns.

Figure 28. Modified Setup Diagram



Related Links

The Intel Quartus Prime Timing Analyzer on page 79
For more information about creating exceptions with multicycle paths.

7.2.6 Metastability

Metastability problems can occur when a signal is transferred between circuitry in unrelated or asynchronous clock domains because the designer cannot guarantee that the signal will meet setup and hold time requirements.

To minimize the failures due to metastability, circuit designers typically use a sequence of registers, also known as a synchronization register chain, or synchronizer, in the destination clock domain to resynchronize the data signals to the new clock domain.

The mean time between failures (MTBF) is an estimate of the average time between instances of failure due to metastability.

The Timing Analyzer analyzes the potential for metastability in your design and can calculate the MTBF for synchronization register chains. The MTBF of the entire design is then estimated based on the synchronization chains it contains.

In addition to reporting synchronization register chains found in the design, the Intel Quartus Prime software also protects these registers from optimizations that might negatively impact MTBF, such as register duplication and logic retiming. The Intel Quartus Prime software can also optimize the MTBF of your design if the MTBF is too low.

Related Links

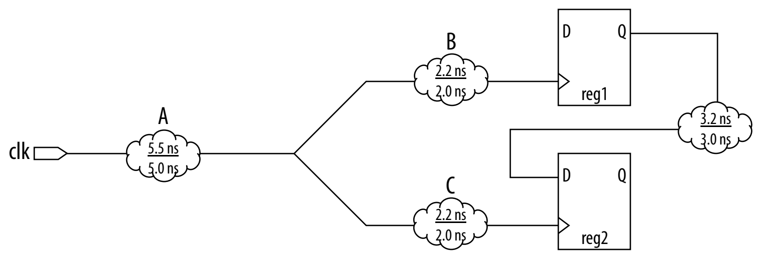
- Understanding Metastability in FPGAs
  For more information about metastability, its effects in FPGAs, and how MTBF is calculated.
- Managing Metastability with the Intel Quartus Prime Software
  For more information about metastability analysis, reporting, and optimization features in the Intel Quartus Prime software.

7.2.7 Common Clock Path Pessimism Removal

Common clock path pessimism removal accounts for the minimum and maximum delay variation associated with common clock paths during static timing analysis by adding the difference between the maximum and minimum delay value of the common clock path to the appropriate slack equation.

Minimum and maximum delay variation can occur when two different delay values are used for the same clock path. For example, in a simple setup analysis, the maximum clock path delay to the source register is used to determine the data arrival time. The minimum clock path delay to the destination register is used to determine the data required time. However, if the clock path to the source register and to the destination register share a common clock path, both the maximum delay and the minimum delay are used to model the common clock path during timing analysis. The use of both the minimum delay and maximum delay results in an overly pessimistic analysis since two different delay values, the maximum and minimum delays, cannot be used to model the same clock path.

Figure 29. Typical Register to Register Path

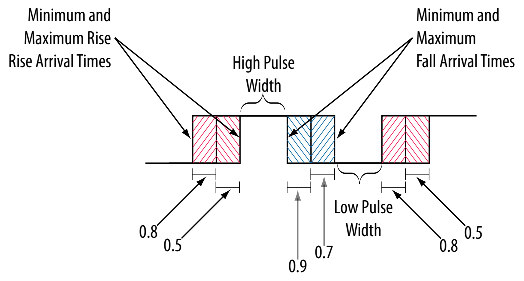


Segment A is the common clock path between `reg1` and `reg2`. The minimum delay is 5.0 ns; the maximum delay is 5.5 ns. The difference between the maximum and minimum delay value equals the common clock path pessimism removal value; in this case, the common clock path pessimism is 0.5 ns. The Timing Analyzer adds the common clock path pessimism removal value to the appropriate slack equation to determine overall slack. Therefore, if the setup slack for the register-to-register path in the example equals 0.7 ns without common clock path pessimism removal, the slack would be 1.2 ns with common clock path pessimism removal.

You can also use common clock path pessimism removal to determine the minimum pulse width of a register. A clock signal must meet a register's minimum pulse width requirement to be recognized by the register. A minimum high time defines the minimum pulse width for a positive-edge triggered register. A minimum low time defines the minimum pulse width for a negative-edge triggered register.

Clock pulses that violate the minimum pulse width of a register prevent data from being latched at the data pin of the register. To calculate the slack of the minimum pulse width, the Timing Analyzer subtracts the required minimum pulse width time from the actual minimum pulse width time. The Timing Analyzer determines the actual minimum pulse width time by the clock requirement you specified for the clock that feeds the clock port of the register. The Timing Analyzer determines the required minimum pulse width time by the maximum rise, minimum rise, maximum fall, and minimum fall times.

Figure 30. Required Minimum Pulse Width time for the High and Low Pulse


Minimum and Maximum Rise Rise Arrival Times
High Pulse Width
Minimum and Maximum Fall Arrival Times
0.8
0.5
0.9
0.7
Low Pulse Width
0.8
0.5

With common clock path pessimism, the minimum pulse width slack can be increased by the smallest value of either the maximum rise time minus the minimum rise time, or the maximum fall time minus the minimum fall time. In the example, the slack value can be increased by 0.2 ns, which is the smallest value between 0.3 ns (0.8 ns – 0.5 ns) and 0.2 ns (0.9 ns – 0.7 ns).

Related Links

Timing Analyzer Page (Settings Dialog Box)
For more information, refer to the Intel Quartus Prime Help.

7.2.8 Clock-As-Data Analysis

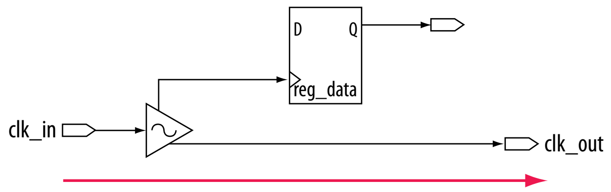
The majority of FPGA designs contain simple connections between any two nodes known as either a data path or a clock path.

A data path is a connection between the output of a synchronous element to the input of another synchronous element.

A clock is a connection to the clock pin of a synchronous element. However, for more complex FPGA designs, such as designs that use source-synchronous interfaces, this simplified view is no longer sufficient. Clock-as-data analysis is performed in circuits with elements such as clock dividers and DDR source-synchronous outputs.

The connection between the input clock port and output clock port can be treated either as a clock path or a data path. A design where the path from port `clk_in` to port `clk_out` is both a clock and a data path. The clock path is from the port `clk_in` to the register `reg_data` clock pin. The data path is from port `clk_in` to the port `clk_out`.

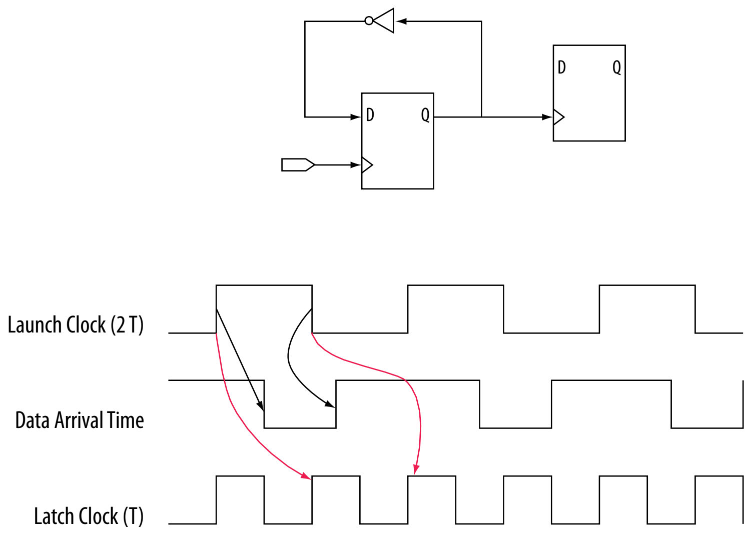
Figure 31. Simplified Source Synchronous Output



With clock-as-data analysis, the Timing Analyzer provides a more accurate analysis of the path based on user constraints. For the clock path analysis, any phase shift associated with the phase-locked loop (PLL) is taken into consideration. For the data path analysis, any phase shift associated with the PLL is taken into consideration rather than ignored.

The clock-as-data analysis also applies to internally generated clock dividers. An internally generated clock divider. In this figure, waveforms are for the inverter feedback path, analyzed during timing analysis. The output of the divider register is used to determine the launch time and the clock port of the register is used to determine the latch time.

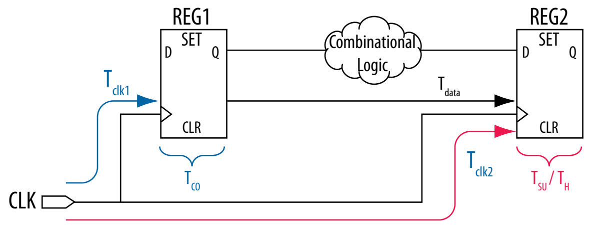
Figure 32. Clock Divider



7.2.9 Multicycle Clock Setup Check and Hold Check Analysis

You can modify the setup and hold relationship when you apply a multicycle exception to a register-to-register path.

Figure 33. Register-to-Register Path

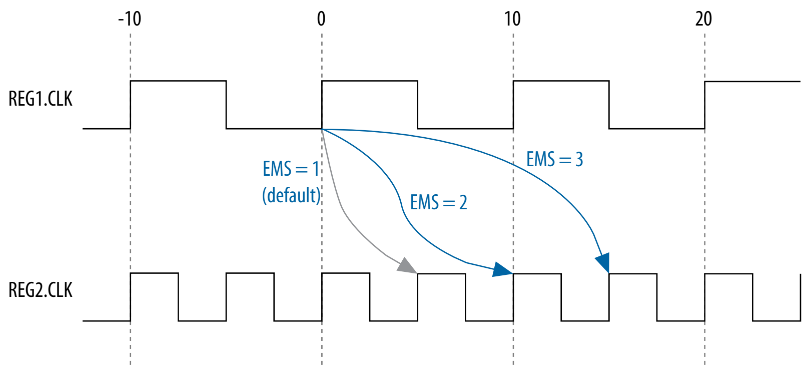


7.2.9.1 Multicycle Clock Setup

The setup relationship is defined as the number of clock periods between the latch edge and the launch edge. By default, the Timing Analyzer performs a single-cycle path analysis, which results in the setup relationship being equal to one clock period (latch edge – launch edge). Applying a multicycle setup assignment, adjusts the setup relationship by the multicycle setup value. The adjustment value may be negative.

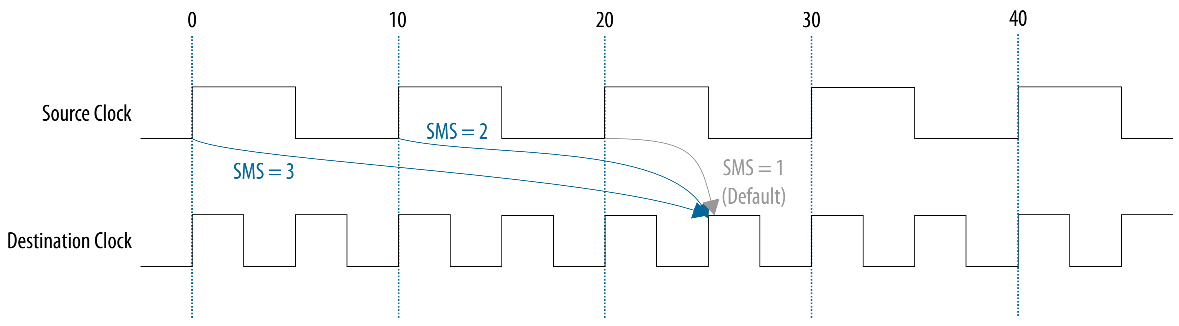
An end multicycle setup assignment modifies the latch edge of the destination clock by moving the latch edge the specified number of clock periods to the right of the determined default latch edge. The following figure shows various values of the end multicycle setup (EMS) assignment and the resulting latch edge.

Figure 34. End Multicycle Setup Values



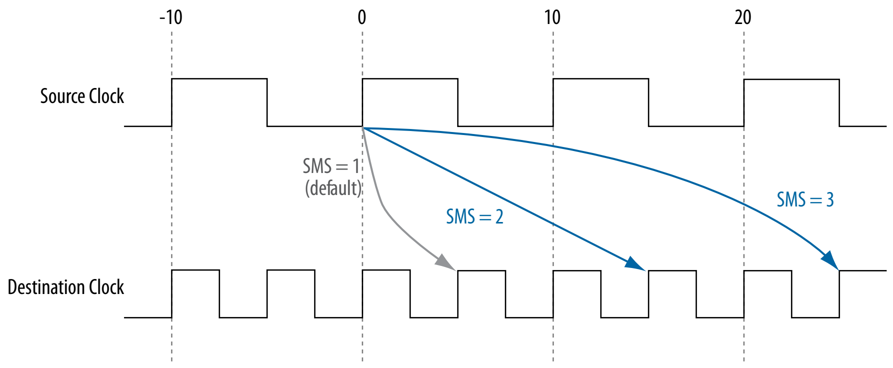
A start multicycle setup assignment modifies the launch edge of the source clock by moving the launch edge the specified number of clock periods to the left of the determined default launch edge. A start multicycle setup (SMS) assignment with various values can result in a specific launch edge.

Figure 35. Start Multicycle Setup Values



The setup relationship reported by the Timing Analyzer for the negative setup relationship.

Figure 36. Start Multicycle Setup Values Reported by the Timing Analyzer



7.2.9.2 Multicycle Clock Hold

The setup relationship is defined as the number of clock periods between the launch edge and the latch edge.

By default, the Timing Analyzer performs a single-cycle path analysis, which results in the hold relationship being equal to one clock period (launch edge – latch edge).When analyzing a path, the Timing Analyzer performs two hold checks. The first hold check determines that the data launched by the current launch edge is not captured by the previous latch edge. The second hold check determines that the data launched by the next launch edge is not captured by the current latch edge. The Timing Analyzer reports only the most restrictive hold check. The Timing Analyzer calculates the hold check by comparing launch and latch edges.

The calculation the Timing Analyzer performs to determine the hold check.

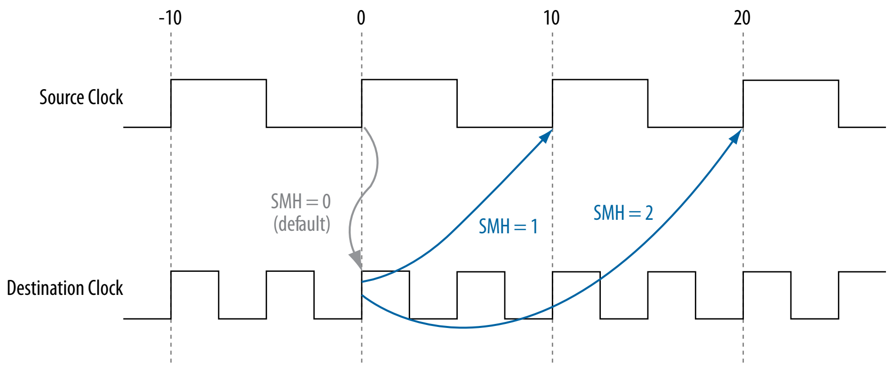
Figure 37. Hold Check

$$\text{hold check 1} = \text{current launch edge} - \text{previous latch edge}$$
$$\text{hold check 2} = \text{next launch edge} - \text{current latch edge}$$

Tip: If a hold check overlaps a setup check, the hold check is ignored.

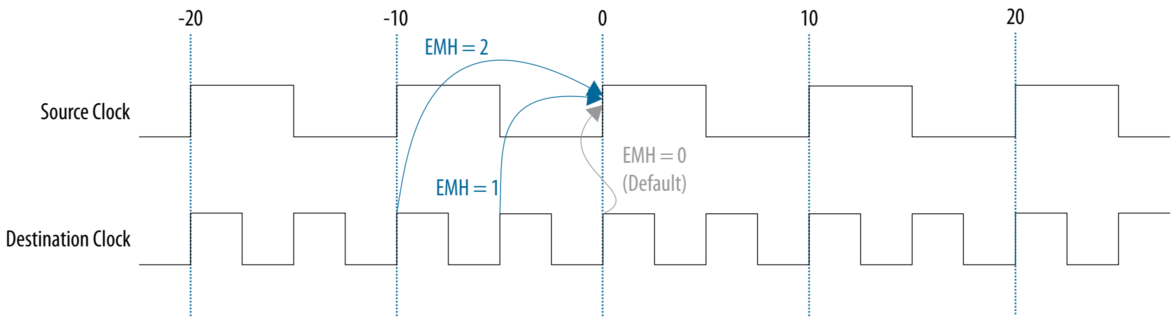
A start multicycle hold assignment modifies the launch edge of the destination clock by moving the latch edge the specified number of clock periods to the right of the determined default launch edge. The following figure shows various values of the start multicycle hold (SMH) assignment and the resulting launch edge.

Figure 38. Start Multicycle Hold Values



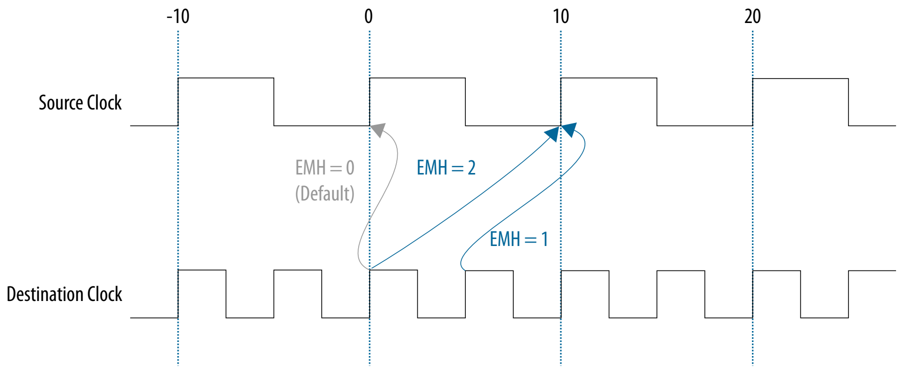
An end multicycle hold assignment modifies the latch edge of the destination clock by moving the latch edge the specific ed number of clock periods to the left of the determined default latch edge. The following figure shows various values of the end multicycle hold (EMH) assignment and the resulting latch edge.

Figure 39. End Multicycle Hold Values



The hold relationship reported by the Timing Analyzer for the negative hold relationship shown in the figure above would look like this:

Figure 40. End Multicycle Hold Values Reported by the Timing Analyzer



7.2.10 Multicorner Analysis

The Timing Analyzer performs multicorner timing analysis to verify your design under a variety of operating conditions—such as voltage, process, and temperature—while performing static timing analysis.

To change the operating conditions or speed grade of the device used for timing analysis, use the `set_operating_conditions` command.

If you specify an operating condition Tcl object, the `-model`, `speed`, `-temperature`, and `-voltage` options are optional. If you do not specify an operating condition Tcl object, the `-model` option is required; the `-speed`, `-temperature`, and `-voltage` options are optional.

Tip: To obtain a list of available operating conditions for the target device, use the `get_available_operating_conditions -all` command.

To ensure that no violations occur under various conditions during the device operation, perform static timing analysis under all available operating conditions.

Table 22. Operating Conditions for Slow and Fast Models

Model	Speed Grade	Voltage	Temperature
Slow	Slowest speed grade in device density	V_{cc} minimum supply [(1)]	Maximum T_J [(1)]
Fast	Fastest speed grade in device density	V_{cc} maximum supply [(1)]	Minimum T_J [(1)]

Note :
1. Refer to the DC & Switching Characteristics chapter of the applicable device Handbook for V_{cc} and T_J. values

In your design, you can set the operating conditions for to the slow timing model, with a voltage of 1100 mV, and temperature of 85° C with the following code:

```
set_operating_conditions -model slow -temperature 85 -voltage 1100
```

You can set the same operating conditions with a Tcl object:

```
set_operating_conditions 3_slow_1100mv_85c
```

The following block of code shows how to use the `set_operating_conditions` command to generate different reports for various operating conditions.

Example 7. Script Excerpt for Analysis of Various Operating Conditions

```
#Specify initial operating conditions
set_operating_conditions -model slow -speed 3 -grade c -temperature 85 -
voltage 1100
#Update the timing netlist with the initial conditions
update_timing_netlist
#Perform reporting
#Change initial operating conditions. Use a temperature of 0C
set_operating_conditions -model slow -speed 3 -grade c -temperature 0 -
voltage 1100
#Update the timing netlist with the new operating condition
update_timing_netlist
#Perform reporting
#Change initial operating conditions. Use a temperature of 0C and a model of
```

```
fast
set_operating_conditions -model fast -speed 3 -grade c -temperature 0 -
voltage 1100
#Update the timing netlist with the new operating condition
update_timing_netlist
#Perform reporting
```

Related Links

- set_operating_conditions
- get_available_operating_conditions
  For more information about the `get_available_operating_conditions` command

7.3 Document Revision History

Table 23. Document Revision History

Date	Version	Changes
2016.05.02	16.0.0	Corrected typo in Fig 6-14: Clock Hold Slack Calculation from Internal Register to Output Port
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
2014.12.15	14.1.0	Moved Multicycle Clock Setup Check and Hold Check Analysis section from the Timing Analyzer chapter.
June 2014	14.0.0	Updated format
June 2012	12.0.0	Added social networking icons, minor text updates
November 2011	11.1.0	Initial release.

Related Links

Documentation Archive
For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

8 The Intel Quartus Prime Timing Analyzer

The Intel Quartus Prime Timing Analyzer is a powerful ASIC-style timing analysis tool that validates the timing performance of all logic in your design using an industry-standard constraint, analysis, and reporting methodology. Use the Timing Analyzer GUI or command-line interface to constrain, analyze, and report results for all timing paths in your design.

This document is organized to allow you to refer to specific subjects relating to the Timing Analyzer and timing analysis. The sections cover the following topics:

Related Links

- Timing Analysis Overview on page 63
- Timing Analyzer Resource Center
- Intel FPGA Technical Training

8.1 Enhanced Timing Analysis for Intel Arria 10 Devices

The Timing Analyzer supports new timing algorithms for the Intel Arria 10 device family which significantly improve the speed of the analysis.

These algorithms are enabled by default for Intel Arria 10 devices, and can be enabled for earlier families with an assignment. The new analysis engine analyzes the timing graph a fixed number of times. Previous Timing Analyzer analysis analyzed the timing graph as many times as there were constraints in your Synopsys Design Constraint (SDC) file.

The new algorithms also support incremental timing analysis, which allows you to modify a single block and re-analyze while maintaining a fully analyzed design.

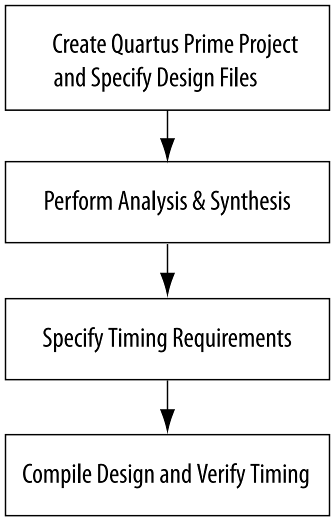
You can turn on the new timing algorithms for use with Arria V, Cyclone V, and Stratix V devices with the following QSF assignment:

```
set_global_assignment -name TIMEQUEST2 ON
```

8.2 Recommended Flow for First Time Users

The Intel Quartus Prime Timing Analyzer performs constraint validation to timing verification as part of the compilation flow. Both the Timing Analyzer and the Fitter use of constraints contained in a Synopsys Design Constraints (`.sdc`) file. The following flow is recommended if you have not created a project and do not have a SDC file with timing constraints for your design.

Figure 41. Design Flow with the Timing Analyzer


Create Quartus Prime Project and Specify Design Files
Perform Analysis & Synthesis
Specify Timing Requirements
Compile Design and Verify Timing

8.2.1 Creating and Setting Up your Design

You must first create your project in the Intel Quartus Prime software. Include all the necessary design files, including any existing Synopsys Design Constraints (`.sdc`) files, also referred to as SDC files, that contain timing constraints for your design. Some reference designs, or Intel FPGA or partner IP cores may already include one or more SDC files.

All SDC files must be added to your project so that your constraints are processed when the Intel Quartus Prime software performs Fitting and Timing Analysis. Typically you must create an SDC file to constrain your design.

Related Links

SDC File Precedence on page 135

8.2.2 Specifying Timing Requirements

Before running timing analysis with the Timing Analyzer, you must specify timing constraints, describe the clock frequency requirements and other characteristics, timing exceptions, and I/O timing requirements of your design. You can use the Timing Analyzer Wizard to enter initial constraints for your design, and then refine timing constraints with the Timing Analyzer GUI.

Both the Timing Analyzer and the Fitter use of constraints contained in a Synopsis Design Constraints (`.sdc`) file.

The constraints in the SDC file are read in sequence. You must first make a constraint before making any references to that constraint. For example, if a generated clock references a base clock, the base clock constraint must be made before the generated clock constraint.

If you are new to timing analysis with the Timing Analyzer, you can use template files included with the Intel Quartus Prime software and the interactive dialog boxes to create your initial SDC file. To use this method, refer to Performing an Initial Analysis and Synthesis.

If you are familiar with timing analysis, you can also create an SDC file in you preferred text editor. Don't forget to include the SDC file in the project when you are finished.

Related Links

- Creating a Constraint File from Intel Quartus Prime Templates with the Intel Quartus Prime Text Editor on page 82
  For more information on using the <keyword keyref="qts-all" /> Text Editor templates for SDC constraints.
- Identifying the Intel Quartus Prime Software Executable from the SDC File on page 156

8.2.2.1 Performing an Initial Analysis and Synthesis

Perform Analysis and Synthesis on your design so that you can find design entry names in the **Node Finder** to simplify creating constraints.

The Intel Quartus Prime software populates an internal database with design element names. You must synthesize your design in order for the Intel Quartus Prime software to assign names to your design elements, for example, pins, nodes, hierarchies, and timing paths.

If you have already compiled your design, you do not need need to perform the synthesis step again, because compiling the design automatically performs synthesis.You can either perform Analysis and Synthesis to create a post-map database, or perform a full compilation to create a post-fit database. Creating a post-map database is faster than a post-fit database, and is sufficient for creating initial timing constraints.

Note: If you are using incremental compilation, you must merge your design partitions after performing Analysis and Synthesis to create a post-map database.

Note: When compiling for the Intel Arria 10 device family, the following commands are required to perform initial synthesis and enable you to use the **Node Finder** to find names in your design:

```
quartus_map <design>
quartus_fit <design> --floorplan
quartus_sta <design> --post_map
```

When compiling for other devices, you can exclude the `quartus_fit <design> --floorplan` step:

```
quartus_map <design>
quartus_sta <design> --post_map
```

8.2.2.2 Creating a Constraint File from Intel Quartus Prime Templates with the Intel Quartus Prime Text Editor

You can create an SDC file from constraint templates in the Intel Quartus Prime software with the Intel Quartus Prime Text Editor, or with your preferred text editor.

1. On the **File** menu, click **New**.
2. In the **New** dialog box, select the **Synopsys Design Constraints File** type from the **Other Files** group. Click **OK**.
3. Right-click in the blank SDC file in the Intel Quartus Prime Text Editor, then click **Insert Constraint**. Choose **Clock Constraint** followed by **Set Clock Groups** since they are the most widely used constraints.
   The Intel Quartus Prime Text Editor displays a dialog box with interactive fields for creating constraints. For example, the **Create Clock** dialog box shows you the waveform for your `create_clock` constraint while you adjust the **Period** and **Rising** and **Falling** waveform edge settings. The actual constraint is displayed in the **SDC command** field. Click **Insert** to use the constraint in your SDC.

   or
4. Click the **Insert Template** button on the text editor menu, or, right-click in the blank SDC file in the Intel Quartus Prime Text Editor, then click **Insert Template**Timing Analyzer.
   a. In the **Insert Template** dialog box, expand the **Timing Analyzer** section, then expand the **SDC Commands** section.
   b. Expand a command category, for example, **Clocks**.
   c. Select a command. The SDC constraint appears in the **Preview** pane.
   d. Click **Insert** to paste the SDC constraint into the blank SDC file you created in step 2.
      This creates a generic constraint for you to edit manually, replacing variables such as clock names, period, rising and falling edges, ports, etc.
5. Repeat as needed with other constraints, or click **Close** to close the **Insert Template** dialog box.

You can now use any of the standard features of the Intel Quartus Prime Text Editor to modify the SDC file, or save the SDC file to edit in a text editor. Your SDC can be saved with the same name as the project, and generally should be stored in the project directory.

Related Links

- Create Clocks Dialog Box
- Set Clock Groups Dialog Box
  For more information on Create Clocks and Set Clock Groups, refer to the Intel Quartus Prime Help.

8.2.3 Performing a Full Compilation

After creating initial timing constraints, compile your design.

During a full compilation, the Fitter uses the Timing Analyzer repeatedly to perform timing analysis with your timing constraints. By default, the Fitter can stop early if it meets your timing requirements, instead of attempting to achieve the maximum performance. You can modify this by changing the Fitter effort settings in the Intel Quartus Prime software.

Related Links

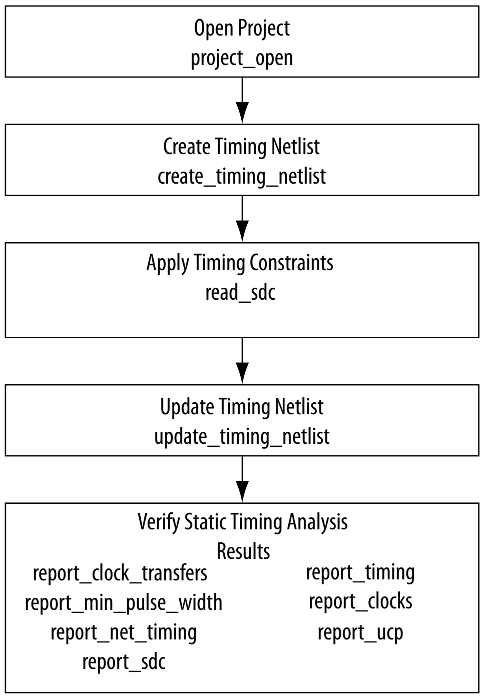
- Analyzing Timing in Designs Compiled in Previous Versions on page 84
  For more information about importing databases compiled in previous versions of the software.
- Fitter Settings Page (Settings Dialog Box)
  For more information about changing Fitter effort, refer to the Intel Quartus Prime Help.

8.2.4 Verifying Timing

The Timing Analyzer examines the timing paths in the design, calculates the propagation delay along each path, checks for timing constraint violations, and reports timing results as positive slack or negative slack. Negative slack indicates a timing violation. If you encounter violations along timing paths, use the timing reports to analyze your design and determine how best to optimize your design. If you modify, remove, or add constraints, you should perform a full compilation again. This iterative process helps resolve timing violations in your design.

There is a recommended flow for constraining and analyzing your design within the Timing Analyzer, and each part of the flow has a corresponding Tcl command.

Figure 42. The Timing Analyzer Flow



8.2.5 Analyzing Timing in Designs Compiled in Previous Versions

Performing a full compilation can be a lengthy process, however, once your design meets timing you can export the design database for later use. This can include operations such as verification of subsequent timing models, or use in a later version of the Intel Quartus Prime software.

When you re-open the project, the Intel Quartus Prime software opens the exported database from the export directory. You can then run Timing Analyzer on the design without having to recompile the project.

To export the database in the previous version of the Intel Quartus Prime software, click **Project ➤ Export Database** and select the export directory to contain the exported database.

To import a database in a later version of the Intel Quartus Prime software, click **File ➤ Open** and select the Intel Quartus Prime Project file (`.qpf`) for the project.

Once you import the database, you can perform any Timing Analyzer functions on the design without recompiling.

8.3 Timing Constraints

Timing analysis in the Intel Quartus Prime software with the Timing Analyzer relies on constraining your design to make it meet your timing requirements. When discussing these constraints, they can be referred to as timing constraints, SDC constraints, or SDC commands interchangeably.

8.3.1 Recommended Starting SDC Constraints

Almost every beginning SDC file should contain the following four commands:

create_clock on page 84

derive_pll_clocks on page 85

derive_clock_uncertainty on page 86

SDC Constraint Creation Summary on page 86

set_clock_groups on page 86

Related Links

Creating a Constraint File from Intel Quartus Prime Templates with the Intel Quartus Prime Text Editor on page 82

8.3.1.1 create_clock

The first statements in a SDC file should be constraints for clocks, for example, constrain the external clocks coming into the FPGA with `create_clock`. An example of the basic syntax is:

```
create_clock -name sys_clk -period 8.0 [get_ports fpga_clk]
```

This command creates a clock called `sys_clk` with an 8ns period and applies it to the port called `fpga_clk`.

Note: Both Tcl files and SDC files are case-sensitive, so make sure references to pins, ports, or nodes, such as `fpga_clk` match the case used in your design.

By default, the clock has a rising edge at time 0ns, and a 50% duty cycle, hence a falling edge at time 4ns. If you require a different duty cycle or to represent an offset, use the `-waveform` option, however, this is seldom necessary.

It is common to create a clock with the same name as the port it is applied to. In the example above, this would be accomplished by:

```
create_clock -name fpga_clk -period 8.0 [get_ports fpga_clk]
```

There are now two unique objects called `fpga_clk`, a port in your design and a clock applied to that port.

Note: In Tcl syntax, square brackets execute the command inside them, so `[get_ports fpga_clk]` executes a command that finds all ports in the design that match `fpga_clk` and returns a collection of them. You can enter the command without using the `get_ports` collection command, as shown in the following example. There are benefits to using collection commands, which are described in "Collection Commands".

```
create_clock -name sys_clk -period 8.0 fpga_clk
```

Repeat this process, using one create_clock command for each known clock coming into your design. Later on you can use **Report Unconstrained Paths** to identify any unconstrained clocks.

Note: Rather than typing constraints, users can enter constraints through the GUI. After launching Timing Analyzer, open the SDC file from Timing Analyzer or Intel Quartus Prime, place the cursor where you want to place the new constraint, and go to **Edit ➤ Insert Constraint**, and choose the constraint.

Warning: Using the **Constraints** menu option in the Timing Analyzer GUI applies constraints directly to the timing database, but makes no entry in the SDC file. An advanced user may find reasons to do this, but if you are new to Timing Analyzer, Intel FPGA recommends entering your constraints directly into your SDC with the **Edit ➤ Insert Constraint** command.

Related Links

Creating Base Clocks on page 90

8.3.1.2 derive_pll_clocks

After the `create_clock` commands add the following command into your SDC file:

```
derive_pll_clocks
```

This command automatically creates a generated clock constraint on each output of the PLLs in your design..

When PLLs are created, you define how each PLL output is configured. Because of this, the Timing Analyzer can automatically constrain them, with the `derive_pll_clocks` command.

This command also creates other constraints. It constrains transceiver clocks. It adds multicycles between LVDS SERDES and user logic.

The **derive_pll_clocks** command prints an Info message to show each generated clock it creates.

If you are new to the Timing Analyzer, you may decide not to use `derive_pll_clocks`, and instead cut-and-paste each `create_generated_clock` assignment into the SDC file. There is nothing wrong with this, since the two are identical. The problem is that when you modifiy a PLL setting, you must remember to change its generated clock in the SDC file. Examples of this type of change include modifying an existing output clock, adding a new PLL output, or making a change to the PLL's hierarchy. Too many designers forget to modify the SDC file and spend time debugging something that `derive_pll_clocks` would have updated automatically.

Related Links

- Creating Base Clocks on page 90
- Deriving PLL Clocks on page 96

8.3.1.3 derive_clock_uncertainty

Add the following command to your SDC file:

```
derive_clock_uncertainty
```

This command calculates clock-to-clock uncertainties within the FPGA, due to characteristics like PLL jitter, clock tree jitter, etc. This should be in all SDC files and the Timing Analyzer generates a warning if this command is not found in your SDC files.

Related Links

Accounting for Clock Effect Characteristics on page 99

8.3.1.4 SDC Constraint Creation Summary

This example shows the SDC file for a sample design with two clocks.

```
create_clock -period 20.00 -name adc_clk [get_ports adc_clk]
create_clock -period 8.00 -name sys_clk [get_ports sys_clk]

derive_pll_clocks

derive_clock_uncertainty
```

8.3.1.5 set_clock_groups

With the constraintsdiscusssed previously, most, if not all, of the clocks in the design are now constrained. In the Timing Analyzer, all clocks are related by default, and you must indicate which clocks are not related. For example, if there are paths between an 8ns clock and 10ns clock, even if the clocks are completely asynchronous, the Timing Analyzer attempts to meet a 2ns setup relationship between these clocks unless you indicate that they are not related. The Timing Analyzer analyzes everything known, rather than assuming that all clocks are unrelated and requiring that you relate them.

The SDC language has a powerful constraint for setting unrelated clocks called `set_clock_groups`. A template for the the typical use of the `set_clock_groups` command is:

```
set_clock_groups -asynchronous -group {<clock1>...<clockn>} ... \
    -group {<clocka>...<clockn>}
```

The `set_clock_groups` command does not actually group clocks. Since the Timing Analyzer assumes all clocks are related by default, all clocks are effectively in one big group. Instead, the `set_clock_groups` command cuts timing between clocks in different groups.

There is no limit to the number of times you can specify a group option with `-group {<group of clocks>}`. When entering constraints through the GUI with **Edit ➤ Insert Constraint**, the **Set Clock Groups** dialog box only permits two clock groups, but this is only a limitation of that dialog box. You can always manually add more into the SDC file.

Any clock not listed in the assignment is related to all clocks. If you forget a clock, the Timing Analyzer acts conservatively and analyzes that clock in context with all other domains to which it connects.

The `set_clock_groups` command requires either the `-asynchronous` or `-exclusive` option. The `-asynchronous` flag means the clocks are both toggling, but not in a way that can synchronously pass data. The `-exclusive` flag means the clocks do not toggle at the same time, and hence are mutually exclusive. An example of this might be a clock multiplexor that has two generated clock assignments on its output. Since only one can toggle at a time, these clocks are `-exclusive`. Timing Analyzer does not currently analyze crosstalk explicitly. Instead, the timing models use extra guard bands to account for any potential crosstalk-induced delays. Timing Analyzer treats the `-asynchronous` and `-exclusive` options the same.

A clock cannot be within multiple `-group` groupings in a single assignment, however, you can have multiple `set_clock_groups` assignments.

Another way to cut timing between clocks is to use `set_false_path`. To cut timing between `sys_clk` and `dsp_clk`, a user might enter:

```
set_false_path -from [get_clocks sys_clk] -to [get_clocks
dsp_clk]

set_false_path -from [get_clocks dsp_clk] -to [sys_clk]
```

This works fine when there are only a few clocks, but quickly grows to a huge number of assignments that are completely unreadable. In a simple design with three PLLs that have multiple outputs, the `set_clock_groups` command can clearly show which clocks are related in less than ten lines, while `set_false_path` may be over 50 lines and be very non-intuitive on what is being cut.

Related Links

- Creating Generated Clocks on page 94
- Relaxing Setup with set_multicyle_path on page 111
- Accounting for a Phase Shift on page 112

8.3.1.5.1 Tips for Writing a set_clock_groups Constraint

Since **derive_pll_clocks** creates many of the clock names, you may not know all of the clock names to use in the clock groups.

A quick way to make this constraint is to use the SDC file you have created so far, with the three basic constraints described in previous topics. Make sure you have added it to your project, then open the Timing Analyzer GUI.

In the **Task** panel of the Timing Analyzer, double-click **Report Clocks**. This reads your existing SDC and applies it to your design, then reports all the clocks. From that report, highlight all of the clock names in the first column, and copy the names.

You have just copied all the clock names in your design in the exact format the Timing Analyzer recognizes. Paste them into your SDC file to make a list of all clock names, one per line..

Format that list into the `set_clock_groups` command by cutting and pasting clock names into appropriate groups. Then enter the following empty template in your SDC file::

```
set_clock_groups -asynchronous -group { \
} \
 -group { \
} \
-group  { \
} \
-group { \
}
```

Cut and paste clocks into groups to define how they're related, adding or removing groups as necessary. Format to make the code readable.

Note: This command can be difficult to read on a single line. Instead, you should make use of the Tcl line continuation character "\". By putting a space after your last character and then "\", the end-of-line character is escaped. (And be careful not to have any whitespace after the escape character, or else the whitespace is read as the character being escaped rather than the end-of-line character).

```
set_clock_groups -asynchronous \
    -group {adc_clk \
       the_adc_pll|altpll_component_autogenerated|pll|clk[0] \
       the_adc_pll|altpll_component_autogenerated|pll|clk[1] \
       the_adc_pll|altpll_component_autogenerated|pll|clk[2] \
    } \
    -group {sys_clk \
       the_system_pll|altpll_component_autogenerated|pll|clk[0] \
       the_system_pll|altpll_component_autogenerated|pll|clk[1] \
    } \
    -group {the_system_pll|altpll_component_autogenerated|pll|clk[2] \
    }
```

Note: The last group has a PLL output `system_pll|..|clk[2]` while the input clock and other PLL outputs are in different groups. If PLLs are used, and the input clock frequency is not related to the frequency of the PLL's outputs, they must be treated asynchronously. Usually most outputs of a PLL are related and hence in the same group, but this is not a requirement, and depends on the requirements of your design.

For designs with complex clocking, writing this constraint can be an iterative process. For example, a design with two DDR3 cores and high-speed transceivers could easily have thirty or more clocks. In those cases, you can just add the clocks you've created. Since clocks not in the command are still related to every clock, you are conservatively grouping what is known. If there are still failing paths in the design between unrelated clock domains, you can start adding in the new clock domains as necessary. In this case, a large number of the clocks won't actually be in the `set_clock_groups` command, since they are either cut in the SDC file for the IP core (such as the SDC files generated by the DDR3 cores), or they only connect to clock domains to which they are related.

For many designs, that is all that's necessary to constrain the core. Some common core constraints that are not be covered in this quick start section that user's do are:

- Add multicycles between registers which can be analyzed at a slower rate than the default analysis, in other words, increasing the time when data can be read, or 'opening the window'. For example, a 10ns clock period has a 10ns setup relationship. If the data changes at a slower rate, or perhaps the registers toggle at a slower rate due to a clock enable, then you should apply a multicycle that relaxes the setup relationship (opens the the window so that valid data can pass). This is a multiple of the clock period, making the setup relationship 20ns, 40ns, etc., while keeping the hold relationship at 0ns. These types of multicycles are generally applied to paths.
- The second common form of multicycle is when the user wants to advance the cycle in which data is read, or 'shift the window'. This generally occurs when your design performs a small phase-shift on a clock. For example, if your design has two 10ns clocks exiting a PLL, but the second clock has a 0.5ns phase-shift, the default setup relationship from the main clock to the phase-shifted clock is 0.5ns and the hold relationship is -9.5ns. It is almost impossible to meet a 0.5ns setup relationship, and most likely you intend the data to transfer in the next window. By adding a multicycle from the main clock to the phase-shifted clock, the setup relationship becomes 10.5ns and the hold relationship becomes 0.5ns. This multicycle is generally applied between clocks and is something the user should think about as soon as they do a small phase-shift on a clock. This type of multicycle is called shifting the window.
- Add a `create_generated_clock` to ripple clocks. When a register's output drives the `clk` port of another register, that is a ripple clock. Clocks do not propagate through registers, so all ripple clocks must have a `create_generated_clock` constraint applied to them for correct analysis. Unconstrained ripple clocks appear in the **Report Unconstrained Paths** report, so they are easily recognized. In general, ripple clocks should be avoided for many reasons, and if possible, a clock enable should be used instead.
- Add a `create_generated_clock` to clock mux outputs. Without this, all clocks propagate through the mux and are related. Timing Analyzer analyze paths downstream from the mux where one clock input feeds the source register and the other clock input feeds the destination, and vice-versa. Although it could be valid, this is usually not preferred behavior. By putting `create_generated_clock` constraints on the mux output, which relates them to the clocks coming into the mux, you can correctly group these clocks with other clocks.

8.3.2 Creating Clocks and Clock Constraints

Clocks specify timing requirements for synchronous transfers and guide the Fitter optimization algorithms to achieve the best possible placement for your design. You must define all clocks and any associated clock characteristics, such as uncertainty or latency. The Timing Analyzer supports SDC commands that accommodate various clocking schemes such as:

- Base clocks
- Virtual clocks
- Multifrequency clocks
- Generated clocks

8.3.2.1 Creating Base Clocks

Base clocks are the primary input clocks to the device. Unlike clocks that are generated in the device (such as an on-chip PLL), base clocks are generated by off-chip oscillators or forwarded from an external device. Define base clocks at the top of your SDC file, because generated clocks and other constraints often reference base clocks. The Timing Analyzer ignores any constraints that reference a clock that has not been defined.

Use the `create_clock` command to create a base clock. Use other constraints, such as those described in *Accounting for Clock Effect Characteristics*, to specify clock characteristics such as uncertainty and latency.

The following examples show the most common uses of the `create_clock` constraint:

create_clock Command

To specify a 100 MHz requirement on a `clk_sys` input clock port you would enter the following in your SDC file:

```
create_clock -period 10 -name clk_sys [get_ports clk_sys]
```

100 MHz Shifted by 90 Degrees Clock Creation

This example creates a 10 ns clock with a 50% duty cycle that is phase shifted by 90 degrees applied to port `clk_sys`. This type of clock definition is most commonly used when the FPGA receives source synchronous, double-rate data that is center-aligned with respect to the clock.

```
create_clock -period 10 -waveform { 2.5 7.5 } [get_ports clk_sys]
```

Two Oscillators Driving the Same Clock Port

You can apply multiple clocks to the same target with the `-add` option. For example, to specify that the same clock input can be driven at two different frequencies, enter the following commands in your SDC file:

```
create_clock -period 10 -name clk_100 [get_ports clk_sys]
create_clock -period 5 -name clk_200 [get_ports clk_sys] -add
```

Although it is not common to have more than two base clocks defined on a port, you can define as many as are appropriate for your design, making sure you specify `-add` for all clocks after the first.

Creating Multifrequency Clocks

You must create a multifrequency clock if your design has more than one clock source feeding a single clock node in your design. The additional clock may act as a low-power clock, with a lower frequency than the primary clock. If your design uses multifrequency clocks, use the `set_clock_groups` command to define clocks that are exclusive.

To create multifrequency clocks, use the `create_clock` command with the `-add` option to create multiple clocks on a clock node. You can create a 10 ns clock applied to clock port `clk`, and then add an additional 15 ns clock to the same clock port. The Timing Analyzer uses both clocks when it performs timing analysis.

```
create_clock –period 10 –name clock_primary –waveform { 0 5 } \
    [get_ports clk]
create_clock –period 15 –name clock_secondary –waveform { 0 7.5 } \
    [get_ports clk] -add
```

Related Links

- Accounting for Clock Effect Characteristics on page 99
- create_clock
- get_ports
  For more information about these commands, refer to Intel Quartus Prime Help.

8.3.2.1.1 Automatically Detecting Clocks and Creating Default Clock Constraints

To automatically create base clocks in your design, use the `derive_clocks` command. The `derive_clocks` command is equivalent to using the `create_clock` command for each register or port feeding the clock pin of a register. The `derive_clocks` command creates clock constraints on ports or registers to ensure every register in your design has a clock constraints, and it applies one period to all base clocks in your design.

You can have the Timing Analyzer create a base clock with a 100 Mhz requirement for unconstrained base clock nodes.

```
derive_clocks -period 10
```

Warning: Do not use the `derive_clocks` command for final timing sign-off; instead, you should create clocks for all clock sources with the `create_clock` and `create_generated_clock` commands. If your design has more than a single clock, the `derive_clocks` command constrains all the clocks to the same specified frequency. To achieve a thorough and realistic analysis of your design's timing requirements, you should make individual clock constraints for all clocks in your design.

If you want to have some base clocks created automatically, you can use the `-create_base_clocks` option to `derive_pll_clocks`. With this option, the `derive_pll_clocks` command automatically creates base clocks for each PLL, based on the input frequency information specified when the PLL was instantiated. The base clocks are named matching the port names. This feature works for simple port-to-PLL connections. Base clocks are not automatically generated for complex PLL connectivity, such as cascaded PLLs. You can also use the command `derive_pll_clocks -create_base_clocks` to create the input clocks for all PLL inputs automatically.

Related Links

derive_clocks
For more information about this command, refer to Intel Quartus Prime Help.

8.3.2.2 Creating Virtual Clocks

A virtual clock is a clock that does not have a real source in the design or that does not interact directly with the design.

To create virtual clocks, use the `create_clock` command with no value specified for the <*targets*> option.

This example defines a 100Mhz virtual clock because no target is specified.

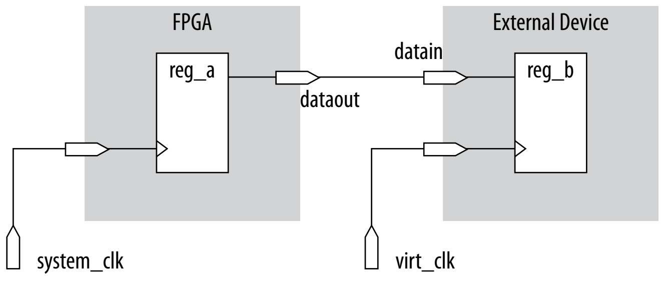
```
create_clock -period 10 -name my_virt_clk
```

I/O Constraints with Virtual Clocks

Virtual clocks are most commonly used in I/O constraints; they represent the clock at the external device connected to the FPGA.

For the output circuit shown in the following figure, you should use a base clock to constrain the circuit in the FPGA, and a virtual clock to represent the clock driving the external device. Examples of the base clock, virtual clock, and output delay constraints for such a circuit are shown below.

Figure 43. Virtual Clock Board Topology



You can create a 10 ns virtual clock named `virt_clk` with a 50% duty cycle where the first rising edge occurs at 0 ns by adding the following code to your SDC file. The virtual clock is then used as the clock source for an output delay constraint.

Example 8. Virtual Clock

```
#create base clock for the design
create_clock -period 5 [get_ports system_clk]
#create the virtual clock for the external register
create_clock -period 10 -name virt_clk
```

```
#set the output delay referencing the virtual clock
set_output_delay -clock virt_clk -max 1.5 [get_ports dataout]
set_output_delay -clock virt_clk -min 0.0 [get_ports dataout]
```

Related Links

- set_input_delay
- set_output_delay
  For more information about these commands, refer to Intel Quartus Prime Help.

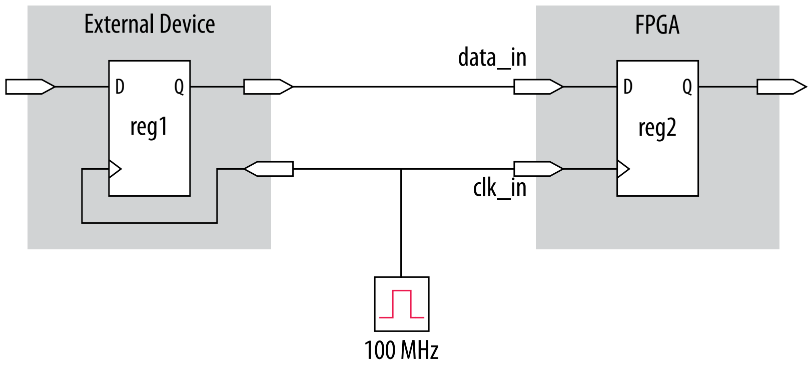
8.3.2.2.1 Example of Specifying an I/O Interface Clock

To specify I/O interface uncertainty, you must create a virtual clock and constrain the input and output ports with the `set_input_delay` and `set_output_delay` commands that reference the virtual clock.

When the `set_input_delay` or `set_output_delay` commands reference a clock port or PLL output, the virtual clock allows the `derive_clock_uncertainty` command to apply separate clock uncertainties for internal clock transfers and I/O interface clock transfers

Create the virtual clock with the same properties as the original clock that is driving the I/O port.

Figure 44. I/O Interface Clock Specifications



Example 9. SDC Commands to Constrain the I/O Interface

```
# Create the base clock for the clock port
create_clock -period 10 -name clk_in [get_ports clk_in]
# Create a virtual clock with the same properties of the base clock
# driving the source register
create_clock -period 10 -name virt_clk_in
# Create the input delay referencing the virtual clock and not the base
# clock
# DO NOT use set_input_delay -clock clk_in <delay value>
# [get_ports data_in]
set_input_delay -clock virt_clk_in <delay value> [get_ports data_in]
```

8.3.2.3 I/O Interface Uncertainty

Virtual clocks are recommended for I/O constraints because they most accurately represent the clocking topology of the design. An additional benefit is that you can specify different uncertainty values on clocks that interface with external I/O ports and clocks that feed register-to-register paths inside the FPGA.

8.3.2.4 Creating Generated Clocks

Define generated clocks on any nodes in your design which modify the properties of a clock signal, including phase, frequency, offset, and duty cycle. Generated clocks are most commonly used on the outputs of PLLs, on register clock dividers, clock muxes, and clocks forwarded to other devices from an FPGA output port, such as source synchronous and memory interfaces. In the SDC file, create generated clocks after the base clocks have been defined. Generated clocks automatically account for all clock delays and clock latency to the generated clock target.

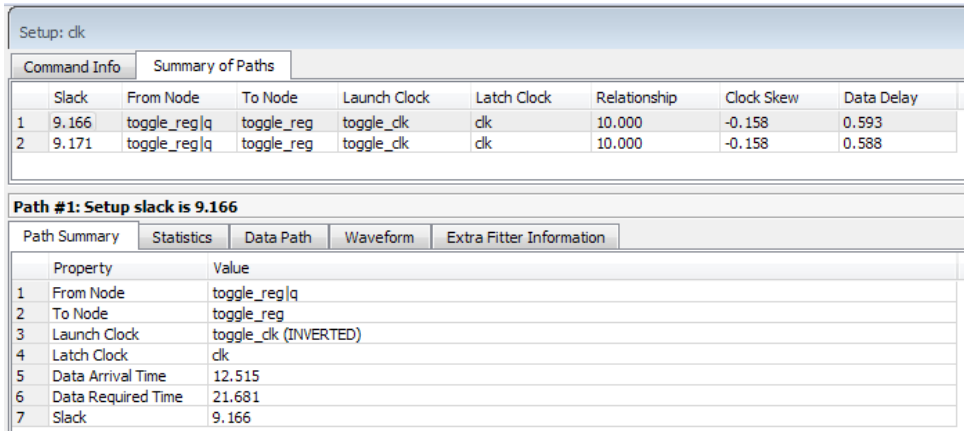
Use the `create_generated_clock` command to constrain generated clocks in your design.

The `-source` option specifies the name of a node in the clock path which is used as reference for your generated clock. The source of the generated clock must be a node in your design netlist and not the name of a previously defined clock. You can use any node name on the clock path between the input clock pin of the target of the generated clock and the target node of its reference clock as the source node. A good practice is to specify the input clock pin of the target node as the source of your new generated clock. That way, the source of the generated clock is decoupled from the naming and hierarchy of its clock source. If you change its clock source, you don't have to edit the generated clock constraint.

If you have multiple base clocks feeding a node that is the source for a generated clock, you must define multiple generated clocks. Each generated clock is associated to one base clock using the `-master_clock` option in each generated clock statement. In some cases, generated clocks are generated with combinational logic. Depending on how your clock-modifying logic is synthesized, the name can change from compile to compile. If the name changes after you write the generated clock constraint, the generated clock is ignored because its target name no longer exists in the design. To avoid this problem, use a synthesis attribute or synthesis assignment to keep the final combinational node of the clock-modifying logic. Then use the kept name in your generated clock constraint. For details on keeping combinational nodes or wires, refer to the *Implement as Output of Logic Cell logic option* topic in Intel Quartus Prime Help.

When a generated clock is created on a node that ultimately feeds the data input of a register, this creates a special case referred to as "clock-as-data". Instances of clock-as-data are treated differently by Timing Analyzer. For example, when clock-as-data is used with DDR, both the rise and the fall of this clock need to be considered since it is a clock, and Timing Analyzer reports both rise and fall. With clock-as-data, the **From Node** is treated as the target of the generated clock, and the **Launch Clock** is treated as the generated clock. In the figure below, the first path is from **toggle_clk (INVERTED)** to **clk**, whereas the second path is from **toggle_clk** to **clk**. The slack in both cases is slightly different due to the difference in rise and fall times along the path; the ~5 ps difference can be seen in the **Data Delay** column. Only the path with the lowest slack value need be considered. This would also be true if this were not a clock-as-data case, but normally Timing Analyzer only reports the worst-case path between the two (rise and fall). In this example, if the generated clock were not defined on the register output, then only one path would be reported and it would be the one with the lowest slack value. If your design targets an Intel Arria 10 device, the enhanced timing algorithms remove all common clock pessimism on paths treated as clock-as-data.

Figure 45. Example of clock-as-data



	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	9.166	toggle_reg|q	toggle_reg	toggle_clk	clk	10.000	-0.158	0.593
2	9.171	toggle_reg|q	toggle_reg	toggle_clk	clk	10.000	-0.158	0.588

Path #1: Setup slack is 9.166

	Property	Value
1	From Node	toggle_reg|q
2	To Node	toggle_reg
3	Launch Clock	toggle_clk (INVERTED)
4	Latch Clock	clk
5	Data Arrival Time	12.515
6	Data Required Time	21.681
7	Slack	9.166

The Timing Analyzer provides the `derive_pll_clocks` command to automatically generate clocks for all PLL clock outputs. The properties of the generated clocks on the PLL outputs match the properties defined for the PLL.

Related Links

- Deriving PLL Clocks on page 96
  For more information about deriving PLL clock outputs.
- Implement as Output of Logic Cell logic option
  For more information on keeping combinational nodes or wires, refer to Intel Quartus Prime Help.
- create_generate_clock
- derive_pll_clocks
- create_generated_clocks
  For information about these commands, refer to Intel Quartus Prime Help.

8.3.2.4.1 Clock Divider Example

A common form of generated clock is a divide-by-two register clock divider. The following constraint creates a half-rate clock on the divide-by-two register.

Figure 46. Clock Divider

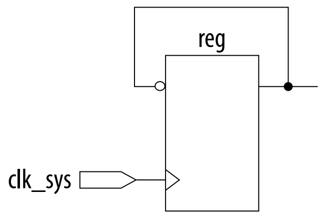
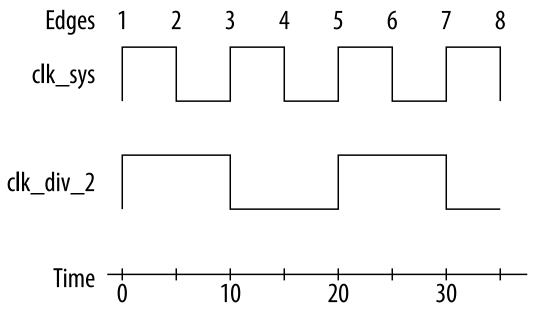


Figure 47. **Clock Divider Waveform**



```
create_clock -period 10 -name clk_sys [get_ports clk_sys]
create_generated_clock -name clk_div_2 -divide_by 2 -source \
    [get_ports clk_sys] [get_pins reg|q]
```

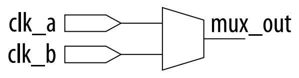
Or in order to have the clock source be the clock pin of the register you can use:

```
create_clock -period 10 -name clk_sys [get_ports clk_sys]
create_generated_clock -name clk_div_2 -divide_by 2 -source \
    [get_pins reg|clk] [get_pins reg|q]
```

8.3.2.4.2 Clock Multiplexor Example

Another common form of generated clock is on the output of a clock mux. One generated clock on the output is requred for each input clock. The SDC example also includes the `set_clock_groups` command to indicate that the two generated clocks can never be active simultaneously in the design, so the Timing Analyzer does not analyze cross-domain paths between the generated clocks on the output of the clock mux.

Figure 48. **Clock Mux**



```
create_clock -name clock_a -period 10 [get_ports clk_a]
create_clock -name clock_b -period 10 [get_ports clk_b]
create_generated_clock -name clock_a_mux -source [get_ports clk_a] \
    [get_pins clk_mux|mux_out]
create_generated_clock -name clock_b_mux -source [get_ports clk_b] \
    [get_pins clk_mux|mux_out] -add
set_clock_groups -exclusive -group clock_a_mux -group clock_b_mux
```

8.3.2.5 Deriving PLL Clocks

Use the `derive_pll_clocks` command to direct the Timing Analyzer to automatically search the timing netlist for all unconstrained PLL output clocks. The `derive_pll_clocks` command detects your current PLL settings and automatically creates generated clocks on the outputs of every PLL by calling the `create_generated_clock` command.

Create Base Clock for PLL input Clock Ports

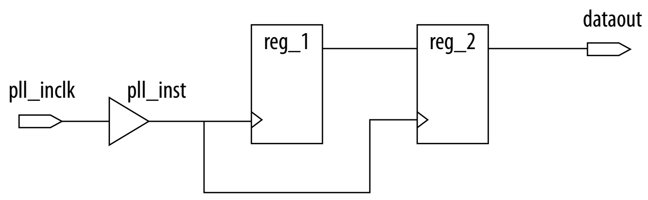
```
create_clock -period 10.0 -name fpga_sys_clk [get_ports fpga_sys_clk] \
    derive_pll_clocks
```

If your design contains transceivers, LVDS transmitters, or LVDS receivers, you must use the `derive_pll_clocks` command. The command automatically constrains this logic in your design and creates timing exceptions for those blocks.

Include the `derive_pll_clocks` command in your SDC file after any `create_clock` command Each time the Timing Analyzer reads your SDC file, the appropriate generate clock is created for each PLL output clock pin. If a clock exists on a PLL output before running `derive_pll_clocks`, the pre-existing clock has precedence, and an auto-generated clock is not created for that PLL output.

A simple PLL design with a register-to-register path.

Figure 49. Simple PLL Design



The Timing Analyzer generates messages when you use the `derive_pll_clocks` command to automatically constrain the PLL for a design similar to the previous image.

Example 10. derive_pll_clocks Command Messages

```
Info:
Info: Deriving PLL Clocks:
Info: create_generated_clock -source pll_inst|altpll_component|pll|inclk[0] -
divide_by 2 -name
pll_inst|altpll_component|pll|clk[0] pll_inst|altpll_component|pll|clk[0]
Info:
```

The input clock pin of the PLL is the node `pll_inst|altpll_component|pll|inclk[0]` which is used for the `-source` option. The name of the output clock of the PLL is the PLL output clock node, `pll_inst|altpll_component|pll|clk[0]`.

If the PLL is in clock switchover mode, multiple clocks are created for the output clock of the PLL; one for the primary input clock (for example, `inclk[0]`), and one for the secondary input clock (for example, `inclk[1]`). You should create exclusive clock groups for the primary and secondary output clocks since they are not active simultaneously.

Related Links

- Creating Clock Groups on page 97
  For more information about creating exclusive clock groups.
- derive_pll_clocks
- Derive PLL Clocks
  For more information about the derive_pll_clocks command.

8.3.2.6 Creating Clock Groups

The Timing Analyzer assumes all clocks are related unless constrained otherwise.

To specify clocks in your design that are exclusive or asynchronous, use the `set_clock_groups` command. The `set_clock_groups` command cuts timing between clocks in different groups, and performs the same analysis regardless of whether you specify `-exclusive` or `-asynchronous`. A group is defined with the `-group` option. The Timing Analyzer excludes the timing paths between clocks for each of the separate groups.

The following tables show examples of various group options for the `set_clock_groups` command.

Table 24. set_clock_groups -group A

Dest\Source	A	B	C	D
A	Analyzed	Cut	Cut	Cut
B	Cut	Analyzed	Analyzed	Analyzed
C	Cut	Analyzed	Analyzed	Analyzed
D	Cut	Analyzed	Analyzed	Analyzed

Table 25. set_clock_groups -group {A B}

Dest\Source	A	B	C	D
A	Analyzed	Analyzed	Cut	Cut
B	Analyzed	Analyzed	Cut	Cut
C	Cut	Cut	Analyzed	Analyzed
D	Cut	Cut	Analyzed	Analyzed

Table 26. set_clock_groups -group A -group B

Dest\Source	A	B	C	D
A	Analyzed	Cut	Cut	Cut
B	Cut	Analyzed	Cut	Cut
C	Cut	Cut	Analyzed	Analyzed
D	Cut	Cut	Analyzed	Analyzed

Table 27. set_clock_groups -group {A C} -group {B D}

Dest\Source	A	B	C	D
A	Analyzed	Cut	Analyzed	Cut
B	Cut	Analyzed	Cut	Analyzed
C	Analyzed	Cut	Analyzed	Cut
D	Cut	Analyzed	Cut	Analyzed

Table 28. set_clock_groups -group {A C D}

Dest\Source	A	B	C	D
A	Analyzed	Cut	Analyzed	Analyzed

continued...

B	Cut	Analyzed	Cut	Cut
C	Analyzed	Cut	Analyzed	Analyzed
D	Analyzed	Cut	Analyzed	Analyzed

Related Links

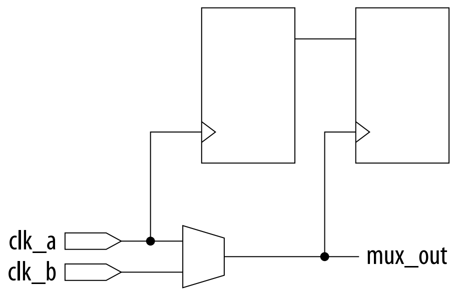
set_clock_groups
For more information about this command, refer to Intel Quartus Prime Help.

8.3.2.6.1 Exclusive Clock Groups

Use the `-exclusive` option to declare that two clocks are mutually exclusive. You may want to declare clocks as mutually exclusive when multiple clocks are created on the same node. This case occurs for multiplexed clocks.

For example, an input port may be clocked by either a 25-MHz or a 50-MHz clock. To constrain this port, create two clocks on the port, and then create clock groups to declare that they do not coexist in the design at the same time. Declaring the clocks as mutually exclusive eliminates clock transfers that are derived between the 25-MHz clock and the 50-MHz clock.

Figure 50. Clock Mux with Synchronous Path Across the Mux



```
create_clock -period 40 -name clk_a [get_ports {port_a}]
create_clock -add -period 20 -name clk_b [get_ports {clk_a}]
set_clock_groups -exclusive -group {clk_a} -group {clk_b}
```

8.3.2.6.2 Asynchronous Clock Groups

Use the `-asynchronous` option to create asynchronous clock groups. Asynchronous clock groups are commonly used to break the timing relationship where data is transfered through a FIFO between clocks running at different rates.

Related Links

set_clock_groups
For more information about this command, refer to Intel Quartus Prime Help.

8.3.2.7 Accounting for Clock Effect Characteristics

The clocks you create with the Timing Analyzer are ideal clocks that do not account for any board effects. You can account for clock effect characteristics with clock latency and clock uncertainty.

8.3.2.7.1 Clock Latency

There are two forms of clock latency, clock source latency and clock network latency. Source latency is the propagation delay from the origin of the clock to the clock definition point (for example, a clock port). Network latency is the propagation delay from a clock definition point to a register's clock pin. The total latency at a register's clock pin is the sum of the source and network latencies in the clock path.

To specify source latency to any clock ports in your design, use the `set_clock_latency` command.

Note: The Timing Analyzer automatically computes network latencies; therefore, you only can characterize source latency with the `set_clock_latency` command. You must use the `-source` option.

Related Links

set_clock_latency
For more information about this command, refer to Intel Quartus Prime Help.

8.3.2.7.2 Clock Uncertainty

When clocks are created, they are ideal and have perfect edges. It is important to add uncertainty to those perfect edges, to mimic clock-level effects like jitter. You should include the `derive_clock_uncertainty` command in your SDC file so that appropriate setup and hold uncertainties are automatically calculated and applied to all clock transfers in your design. If you don't include the command, the Timing Analyzer performs it anyway; it is a critical part of constraining your design correctly.

The Timing Analyzer subtracts setup uncertainty from the data required time for each applicable path and adds the hold uncertainty to the data required time for each applicable path. This slightly reduces the setup and hold slack on each path.

The Timing Analyzer accounts for uncertainty clock effects for three types of clock-to-clock transfers; intraclock transfers, interclock transfers, and I/O interface clock transfers.

- Intraclock transfers occur when the register-to-register transfer takes place in the device and the source and destination clocks come from the same PLL output pin or clock port.
- Interclock transfers occur when a register-to-register transfer takes place in the core of the device and the source and destination clocks come from a different PLL output pin or clock port.
- I/O interface clock transfers occur when data transfers from an I/O port to the core of the device or from the core of the device to the I/O port.

To manually specify clock uncertainty, use the `set_clock_uncertainty` command. You can specify the uncertainty separately for setup and hold. You can also specify separate values for rising and falling clock transitions, although this is not commonly used. You can override the value that was automatically applied by the `derive_clock_uncertainty` command, or you can add to it.

The `derive_clock_uncertainty` command accounts for PLL clock jitter if the clock jitter on the input to a PLL is within the input jitter specification for PLL's in the specified device. If the input clock jitter for the PLL exceeds the specification, you should add additional uncertainty to your PLL output clocks to account for excess jitter with the `set_clock_uncertainty -add` command. Refer to the device handbook for your device for jitter specifications.

Another example is to use `set_clock_uncertainty -add` to add uncertainty to account for peak-to-peak jitter from a board when the jitter exceeds the jitter specification for that device. In this case you would add uncertainty to both setup and hold equal to 1/2 the jitter value:

```
set_clock_uncertainty –setup –to <clock name>  \
    –setup –add <p2p jitter/2>
```

```
set_clock_uncertainty –hold –enable_same_physical_edge –to <clock name> \
    –add <p2p jitter/2>
```

There is a complex set of precedence rules for how the Timing Analyzer applies values from `derive_clock_uncertainty` and `set_clock_uncertainty`, which depend on the order the commands appear in your SDC files, and various options used with the commands. The Help topics referred to below contain complete descriptions of these rules. These precedence rules are much simpler to understand and implement if you follow these recommendations:

- If you want to assign your own clock uncertainty values to any clock transfers, the best practice is to put your `set_clock_uncertainty` exceptions after the `derive_clock_uncertainty` command in your SDC file.
- When you use the `-add` option for `set_clock_uncertainty`, the value you specify is added to the value from `derive_clock_uncertainty`. If you don't specify `-add`, the value you specify replaces the value from `derive_clock_uncertainty`.

Related Links

- set_clock_uncertainty
- derive_clock_uncertainty
- remove_clock_uncertainty
  For more information about these commands, refer to Intel Quartus Prime Help.

8.3.3 Creating I/O Requirements

The Timing Analyzer reviews setup and hold relationships for designs in which an external source interacts with a register internal to the design. The Timing Analyzer supports input and output external delay modeling with the `set_input_delay` and `set_output_delay` commands. You can specify the clock and minimum and maximum arrival times relative to the clock.

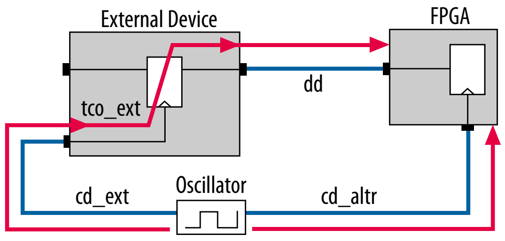
You must specify timing requirements, including internal and external timing requirements, before you fully analyze a design. With external timing requirements specified, the Timing Analyzer verifies the I/O interface, or periphery of the device, against any system specification.

8.3.3.1 Input Constraints

Input constraints allow you to specify all the external delays feeding into the device. Specify input requirements for all input ports in your design.

You can use the `set_input_delay` command to specify external input delay requirements. Use the `-clock` option to reference a virtual clock. Using a virtual clock allows the Timing Analyzer to correctly derive clock uncertainties for interclock and intraclock transfers. The virtual clock defines the launching clock for the input port. The Timing Analyzer automatically determines the latching clock inside the device that captures the input data, because all clocks in the device are defined.

Figure 51. Input Delay



The calculation the Timing Analyzer performs to determine the typical input delay.

Figure 52. Input Delay Calculation

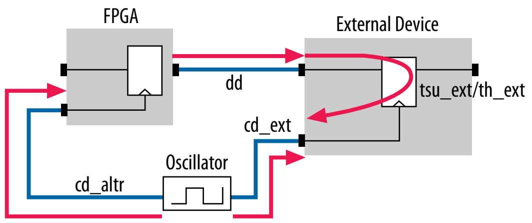
$$\text{input delay}_{MAX} = (cd\_ext_{MAX} - cd\_altr_{MIN}) + tco\_ext_{MAX} + dd_{MAX}$$
$$\text{input delay}_{MIN} = (cd\_ext_{MIN} - cd\_altr_{MAX}) + tco\_ext_{MIN} + dd_{MIN}$$

8.3.3.2 Output Constraints

Output constraints allow you to specify all external delays from the device for all output ports in your design.

You can use the `set_output_delay` command to specify external output delay requirements. Use the `-clock` option to reference a virtual clock. The virtual clock defines the latching clock for the output port. The Timing Analyzer automatically determines the launching clock inside the device that launches the output data, because all clocks in the device are defined. The following figure is an example of an output delay referencing a virtual clock.

Figure 53. Output Delay



The calculation the Timing Analyzer performs to determine the typical out delay.

Figure 54. Output Delay Calculation

$$\text{output delay}_{MAX} = dd_{MAX} + tsu\_ext + (cd\_altr_{MAX} - cd\_ext_{MIN})$$
$$\text{output delay}_{MIN} = (dd_{MIN} - th\_ext + (cd\_altr_{MIN} - cd\_ext_{MAX}))$$

Related Links

- set_intput_delay
- set_output_delay
  For more information about these commands, refer to Intel Quartus Prime Help.

8.3.4 Creating Delay and Skew Constraints

The Timing Analyzer supports the Synopsys Design Constraint format for constraining timing for the ports in your design. These constraints allow the Timing Analyzer to perform a system static timing analysis that includes not only the device internal timing, but also any external device timing and board timing parameters.

8.3.4.1 Advanced I/O Timing and Board Trace Model Delay

The Timing Analyzer can use advanced I/O timing and board trace model assignments to model I/O buffer delays in your design.

If you change any advanced I/O timing settings or board trace model assignments, recompile your design before you analyze timing, or use the `-force_dat` option to force delay annotation when you create a timing netlist.

Example 11. Forcing Delay Annotation

```
create_timing_netlist -force_dat
```

Related Links

- Using Advanced I/O Timing
- I/O Management
  For more information about advanced I/O timing.

8.3.4.2 Maximum Skew

To specify the maximum path-based skew requirements for registers and ports in the design and report the results of maximum skew analysis, use the `set_max_skew` command in conjunction with the `report_max_skew` command.

Use the `set_max_skew` constraint to perform maximum allowable skew analysis between sets of registers or ports. In order to constrain skew across multiple paths, all such paths must be defined within a single `set_max_skew` constraint. `set_max_skew` timing constraint is not affected by **set_max_delay**, `set_min_delay`, and `set_multicycle_path` but it does obey `set_false_path` and `set_clock_groups`. If your design targets an Intel Arria 10 device, skew constraints are not affected by `set_clock_groups`.

Table 29. set_max_skew Options

Arguments	Description
`-h | -help`	Short help
`-long_help`	Long help with examples and possible return values
`-exclude <Tcl list>`	A Tcl list of parameters to exclude during skew analysis. This list can include one or more of the following: `utsu`, `uth`, `utco`, `from_clock`, `to_clock`, `clock_uncertainty`, `ccpp`, `input_delay`, `output_delay`, `odv`. *Note:* Not supported for Intel Arria 10 devices.
`-fall_from_clock <names>`	Valid source clocks (string patterns are matched using Tcl string matching)
`-fall_to_clock <names>`	Valid destination clocks (string patterns are matched using Tcl string matching)
`-from <names>`(1)	Valid sources (string patterns are matched using Tcl string matching
`-from_clock <names>`	Valid source clocks (string patterns are matched using Tcl string matching)
`-get_skew_value_from_clock_period <src_clock_period|dst_clock_period|min_clock_period>`	Option to interpret skew constraint as a multiple of the clock period
`-include <Tcl list>`	Tcl list of parameters to include during skew analysis. This list can include one or more of the following: `utsu`, `uth`, `utco`, `from_clock`, `to_clock`, `clock_uncertainty`, `ccpp`, `input_delay`, `output_delay`, `odv`. *Note:* Not supported for Intel Arria 10 devices.
`-rise_from_clock <names>`	Valid source clocks (string patterns are matched using Tcl string matching)
`-rise_to_clock <names>`	Valid destination clocks (string patterns are matched using Tcl string matching)
`-skew_value_multiplier <multiplier>`	Value by which the clock period should be multiplied to compute skew requirement.
`-to <names>`(1)	Valid destinations (string patterns are matched using Tcl string matching)
`-to_clock <names>`	Valid destination clocks (string patterns are matched using Tcl string matching)
`<skew>`	Required skew

Applying maximum skew constraints between clocks applies the constraint from all register or ports driven by the clock specified with the `-from` option to all registers or ports driven by the clock specified with the `-to` option.

Use the `-include` and `-exclude` options to include or exclude one or more of the following: register micro parameters (`utsu`, `uth`, `utco`), clock arrival times (`from_clock`, `to_clock`), clock uncertainty (`clock_uncertainty`), common clock path pessimism removal (`ccpp`), input and output delays (`input_delay`,

(1) Legal values for the -from and -to options are collections of clocks, registers, ports, pins, cells or partitions in a design.

`output_delay`) and on-die variation (`odv`). Max skew analysis can include data arrival times, clock arrival times, register micro parameters, clock uncertainty, on-die variation and ccpp removal. Among these, only ccpp removal is disabled during the Fitter by default. When `-include` is used, those in the inclusion list are added to the default analysis. Similarly, when `-exclude` is used, those in the exclusion list are excluded from the default analysis. When both the `-include` and `-exclude` options specify the same parameter, that parameter is excluded.

Note: If your design targets an Intel Arria 10 device, `-exclude` and `-include` are not supported.

Use `-get_skew_value_from_clock_period` to set the skew as a multiple of the launching or latching clock period, or whichever of the two has a smaller period. If this option is used, then `-skew_value_multiplier` must also be set, and the positional skew option may not be set. If the set of skew paths is clocked by more than one clock, Timing Analyzer uses the one with smallest period to compute the skew constraint.

When this constraint is used, results of max skew analysis are displayed in the Report Max Skew (report_max_skew) report from the Timing Analyzer. Since skew is defined between two or more paths, no results are displayed if the `-from`/`-from_clock` and `-to`/`-to_clock` filters satisfy less than two paths.

Related Links

- set_max_skew
- report_max_skew
  For more information about these commands, refer to Intel Quartus Prime Help.

8.3.4.3 Net Delay

Use the `set_net_delay` command to set the net delays and perform minimum or maximum timing analysis across nets.

The `-from` and `-to` options can be string patterns or pin, port, register, or net collections. When pin or net collection is used, the collection should include output pins or nets.

Table 30. set_net_delay Options

Arguments	Description
`-h | -help`	Short help
`-long_help`	Long help with examples and possible return values
`-from <names>`	Valid source pins, ports, registers or nets(string patterns are matched using Tcl string matching)
`-get_value_from_clock_period <src_clock_period|dst_clock_period|min_clock_period|max_clock_period>`	Option to interpret net delay constraint as a multiple of the clock period.
`-max`	Specifies maximum delay
`-min`	Specifies minimum delay

continued...

Arguments	Description
`-to <names>`[2]	Valid destination pins, ports, registers or nets (string patterns are matched using Tcl string matching)
`-value_multiplier <multiplier>`	Value by which the clock period should be multiplied to compute net delay requirement.
`<delay>`	Required delay

When you use the -min option, slack is calculated by looking at the minimum delay on the edge. If you use -max option, slack is calculated with the maximum edge delay.

Use `-get_skew_value_from_clock_period` to set the net delay requirement as a multiple of the launching or latching clock period, or whichever of the two has a smaller or larger period. If this option is used, then `-value_multiplier` must also be set, and the positional delay option may not be set. If the set of nets is clocked by more than one clock, Timing Analyzer uses the net with smallest period to compute the constraint for a `-max` constraint, and the largest period for a `-min` constraint. If there are no clocks clocking the endpoints of the net (e.g. if the endpoints of the nets are not registers or constraint ports), then the net delay constraint is ignored.

Related Links

- set_net_delay
- report_net_delay
  For more information about these commands, refer to Intel Quartus Prime Help.

8.3.4.4 Using create_timing_netlist

You can onfigure or load the timing netlist that the Timing Analyzer uses to calculate path delay data.

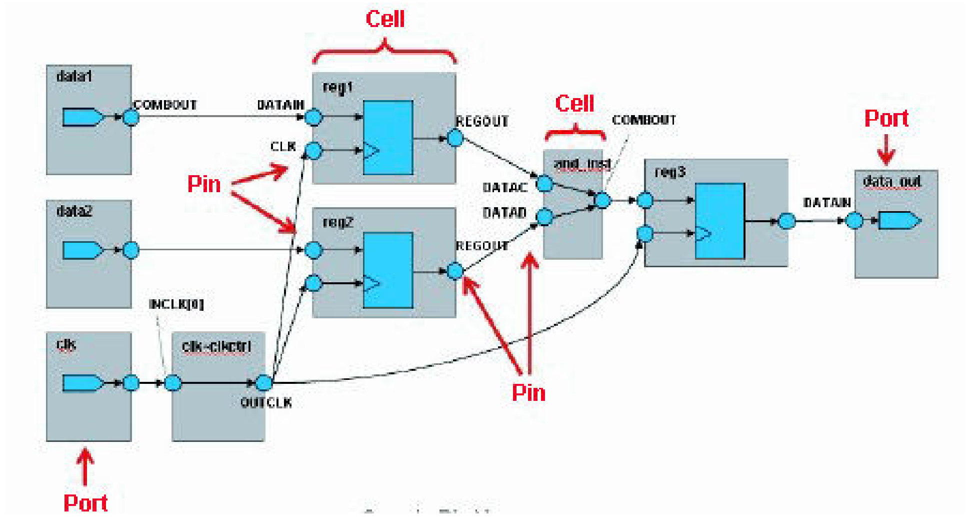
Your design should have a timing netlist before running the Timing Analyzer . You can use the **Create Timing Netlist** dialog box or the **Create Timing Netlist** command in the **Tasks** pane. The command also generates Advanced I/O Timing reports if you turned on **Enable Advanced I/O Timing** in the **Timing Analyzer** page of the **Settings** dialog box.

Note: The timing netlist created is based on the initial configuration of the design. Any configuration changes done by the design after the device enters user mode, for example, dynamic transceiver reconfiguration, are not reflected in the timing netlist. This applies to all device families except transceivers on Intel Arria 10 devices with the Multiple Reconfiguration Profiles feature.

The following diagram shows how the Timing Analyzer interprets and classifies timing netlist data for a sample design.

[2] If the -to option is unused or if the `-to` filter is a wildcard ("*") character, all the output pins and registers on timing netlist became valid destination points.

Figure 55. How Timing Analyzer Interprets the Timing Netlist



8.3.5 Creating Timing Exceptions

Timing exceptions in the Timing Analyzer provide a way to modify the default timing analysis behavior to match the analysis required by your design. Specify timing exceptions after clocks and input and output delay constraints because timing exceptions can modify the default analysis.

8.3.5.1 Precedence

If the same clock or node names occur in multiple timing exceptions, the following order of precedence applies:

1. False path
2. Minimum delays and maximum delays
3. Multicycle path

The false path timing exception has the highest precedence. Within each category, assignments to individual nodes have precedence over assignments to clocks. For exceptions of the same type:

- having a `-from <node>` has the highest priority
- followed by the `-to <node>`
- then the `-thru <node>`
- then `-from <clock>`
- and lastly, a `-to <clock>`

An asterisk wildcard (`*`) for any of these options applies the same precedence as not specifying the option at all. For example, `-from a -to *` is treated identically to `-from a` for as regards precedence.

Precedence example:

1. `set_max_delay 1 -from x -to y`
2. `set_max_delay 2 -from x`
3. `set_max_delay 3 -to y`

The first exception would have higher priority than either of the other two, since it specifies a `-from` (while #3 doesn't) and sepecifies a `-to` (while #2 doesn't). In the absence of the first exception, the second exception would have higher priority than the third, since it specifies a `-from`, which the third does not. Finally, the remaining order of precedence for additional exceptions is order-dependent, such that the assignments most recently created overwrite, or partially overwrite, earlier assignments.

`set_net_delay` or `set_max_skew` exceptions are analyzed independently of minimum or maximum delays, or multicycle path constratints.

- The `set_net_delay` exception applies regardless the existance of a `set_false_path` exception, or `set_clock_group` exception, on the same nodes.
- The `set_max_skew` exception applies regardless of any `set_clock_group` exception on the same nodes, but a `set_false_path` exception overrides a `set_max_skew` exception.

8.3.5.2 False Paths

Specifying a false path in your design removes the path from timing analysis.

Use the `set_false_path` command to specify false paths in your design. You can specify either a point-to-point or clock-to-clock path as a false path. For example, a path you should specify as false path is a static configuration register that is written once during power-up initialization, but does not change state again. Although signals from static configuration registers often cross clock domains, you may not want to make false path exceptions to a clock-to-clock path, because some data may transfer across clock domains. However, you can selectively make false path exceptions from the static configuration register to all endpoints.

To make false path exceptions from all registers beginning with A to all registers beginning with B, use the following code in your SDC file.

```
set_false_path -from [get_pins A*] -to [get_pins B*]
```

The Timing Analyzer assumes all clocks are related unless you specify otherwise. Clock groups are a more efficient way to make false path exceptions between clocks, compared to writing multiple `set_false_path` exceptions between every clock transfer you want to eliminate.

Related Links

- Creating Clock Groups on page 97
  For more information about creating exclusive clock groups.
- set_false_path
  For more information about this command, refer to Intel Quartus Prime Help.

8.3.5.3 Minimum and Maximum Delays

To specify an absolute minimum or maximum delay for a path, use the `set_min_delay` command or the `set_max_delay` commands, respectively. Specifying minimum and maximum delay directly overwrites existing setup and hold relationships with the minimum and maximum values.

Use the `set_max_delay` and `set_min_delay` commands to create constraints for asynchronous signals that do not have a specific clock relationship in your design, but require a minimum and maximum path delay. You can create minimum and maximum delay exceptions for port-to-port paths through the device without a register stage in the path. If you use minimum and maximum delay exceptions to constrain the path delay, specify both the minimum and maximum delay of the path; do not constrain only the minimum or maximum value.

If the source or destination node is clocked, the Timing Analyzer takes into account the clock paths, allowing more or less delay on the data path. If the source or destination node has an input or output delay, that delay is also included in the minimum or maximum delay check.

If you specify a minimum or maximum delay between timing nodes, the delay applies only to the path between the two nodes. If you specify a minimum or maximum delay for a clock, the delay applies to all paths where the source node or destination node is clocked by the clock.

You can create a minimum or maximum delay exception for an output port that does not have an output delay constraint. You cannot report timing for the paths associated with the output port; however, the Timing Analyzer reports any slack for the path in the setup summary and hold summary reports. Because there is no clock associated with the output port, no clock is reported for timing paths associated with the output port.

Note: To report timing with clock filters for output paths with minimum and maximum delay constraints, you can set the output delay for the output port with a value of zero. You can use an existing clock from the design or a virtual clock as the clock reference.

Related Links

- set_max_delay
- set_min_delay
  For more information about these commands, refer to Intel Quartus Prime Help.

8.3.5.4 Delay Annotation

To modify the default delay values used during timing analysis, use the `set_annotated_delay` and `set_timing_derate` commands. You must update the timing netlist to see the results of these commands

To specify different operating conditions in a single SDC file, rather than having multiple SDC files that specify different operating conditions, use the `set_annotated_delay -operating_conditions` command.

Related Links

- set_timing_derate

- set_annotated_delay
  For more information about these commands, refer to the Intel Quartus Prime Help.

8.3.5.5 Multicycle Paths

By default, the Timing Analyzerr performs a single-cycle analysis, which is the most restrictive type of analysis. When analyzing a path, the setup launch and latch edge times are determined by finding the closest two active edges in the respective waveforms.

For a hold analysis, the timing analyzer analyzes the path against two timing conditions for every possible setup relationship, not just the worst-case setup relationship. Therefore, the hold launch and latch times may be completely unrelated to the setup launch and latch edges. The Timing Analyzer does not report negative setup or hold relationships. When either a negative setup or a negative hold relationship is calculated, the Timing Analyzer moves both the launch and latch edges such that the setup and hold relationship becomes positive.

A multicycle constraint adjusts setup or hold relationships by the specified number of clock cycles based on the source (`-start`) or destination (`-end`) clock. An end setup multicycle constraint of 2 extends the worst-case setup latch edge by one destination clock period. If `-start` and `-end` values are not specified, the default constraint is `-end`.

Hold multicycle constraints are based on the default hold position (the default value is 0). An end hold multicycle constraint of 1 effectively subtracts one destination clock period from the default hold latch edge.

When the objects are timing nodes, the multicycle constraint only applies to the path between the two nodes. When an object is a clock, the multicycle constraint applies to all paths where the source node (`-from`) or destination node (`-to`) is clocked by the clock. When you adjust a setup relationship with a multicycle constraint, the hold relationship is adjusted automatically.

You can use Timing Analyzer commands to modify either the launch or latch edge times that the uses to determine a setup relationship or hold relationship.

Table 31. Commands to Modify Edge Times

Command	Description of Modification
`set_multicycle_path -setup -end <value>`	Latch edge time of the setup relationship
`set_multicycle_path -setup -start<value>`	Launch edge time of the setup relationship
`set_multicycle_path -hold -end <value>`	Latch edge time of the hold relationship
`set_multicycle_path -hold -start <value>`	Launch edge time of the hold relationship

8.3.5.6 Common Multicycle Variations

Multicycle exceptions adjust the timing requirements for a register-to-register path, allowing the Fitter to optimally place and route a design in a device. Multicycle exceptions also can reduce compilation time and improve the quality of results, and can be used to change timing requirements. Two common multicycle variations are relaxing setup to allow a slower data transfer rate, and altering the setup to account for a phase shift.

8.3.5.6.1 Relaxing Setup with set_multicycle_path

A common type of multicycle exception occurs when the data transfer rate is slower than the clock cycle. Relaxing the setup relationship opens the window when data is accepted as valid.

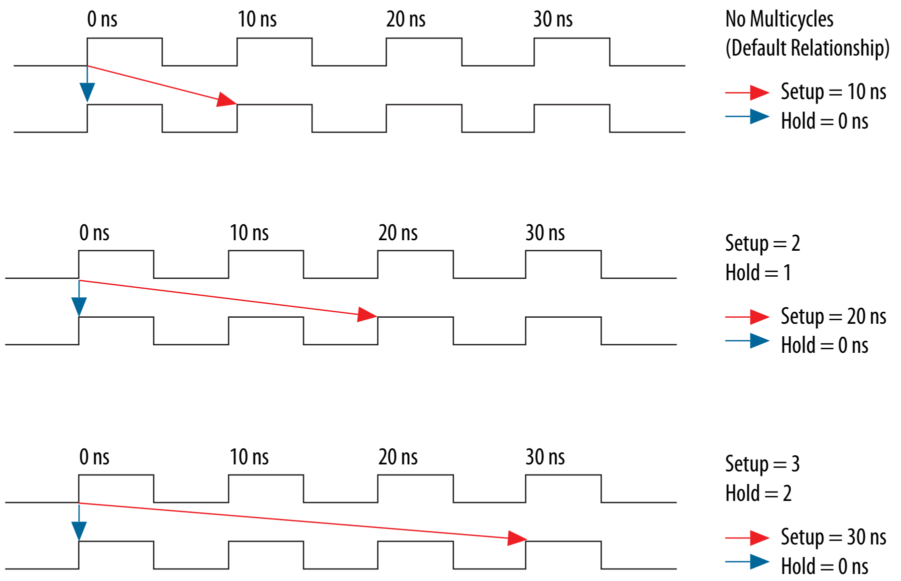
In this example, the source clock has a period of 10 ns, but a group of registers are enabled by a toggling clock, so they only toggle every other cycle. Since they are fed by a 10 ns clock, the Timing Analyzer reports a set up of 10 ns and a hold of 0 ns, However, since the data is transferring every other cycle, the relationships should be analyzed as if the clock were operating at 20 ns, which would result in a setup of 20 ns, while the hold remains 0 ns, in essence, extending the window of time when the data can be recognized.

The following pair of multicycle assignments relax the setup relationship by specifying the `-setup` value of N and the `-hold` value as N-1. You must specify the hold relationship with a `-hold` assignment to prevent a positive hold requirement.

Relaxing Setup while Maintaining Hold

```
set_multicycle_path -setup -from src_reg* -to dst_reg* 2
set_multicycle_path -hold -from src_reg* -to dst_reg* 1
```

Figure 56. Relaxing Setup by Multiple Cycles



This pattern can be extended to create larger setup relationships in order to ease timing closure requirements. A common use for this exception is when writing to asynchronous RAM across an I/O interface. The delay between address, data, and a write enable may be several cycles. A multicycle exception to I/O ports can allow extra time for the address and data to resolve before the enable occurs.

You can relax the setup by three cycles with the following code in your SDC file.

Three Cycle I/O Interface Exception

```
set_multicycle_path -setup -to [get_ports {SRAM_ADD[*] SRAM_DATA[*]} 3
set_multicycle_path -hold -to [get_ports {SRAM_ADD[*] SRAM_DATA[*]} 2
```

8.3.5.6.2 Accounting for a Phase Shift

In this example, the design contains a PLL that performs a phase-shift on a clock whose domain exchanges data with domains that do not experience the phase shift. A common example is when the destination clock is phase-shifted forward and the source clock is not, the default setup relationship becomes that phase-shift, thus shifting the window when data is accepted as valid.

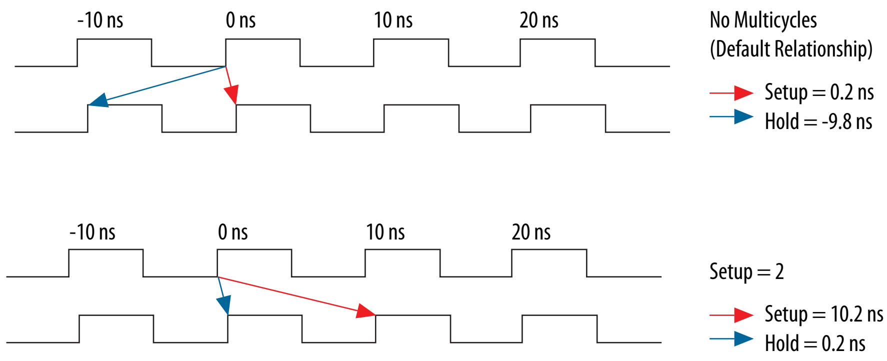
For example, the following code is a circumstance where a PLL phase-shifts one output forward by a small amount, in this case 0.2 ns.

Cross Domain Phase-Shift

```
create_generated_clock -source pll|inclk[0] -name pll|clk[0] pll|clk[0]
create_generated_clock -source pll|inclk[0] -name pll|clk[1] -phase 30 pll|
clk[1]
```

The default setup relationship for this phase-shift is 0.2 ns, shown in Figure A, creating a scenario where the hold relationship is negative, which makes achieving timing closure nearly impossible.

Figure 57. Phase-Shifted Setup and Hold



Adding the following constraint in your SDC allows the data to transfer to the following edge.

```
set_multicycle_path -setup -from [get_clocks clk_a] -to [get_clocks clk_b] 2
```

The hold relationship is derived from the setup relationship, making a multicyle hold constraint unnecessary.

Related Links

- Same Frequency Clocks with Destination Clock Offset on page 121
  Refer to this topic for a more complete example.
- set_multicycle_path
  For more information about this command, refer to the Intel Quartus Prime Help.

8.3.5.7 Examples of Basic Multicycle Exceptions

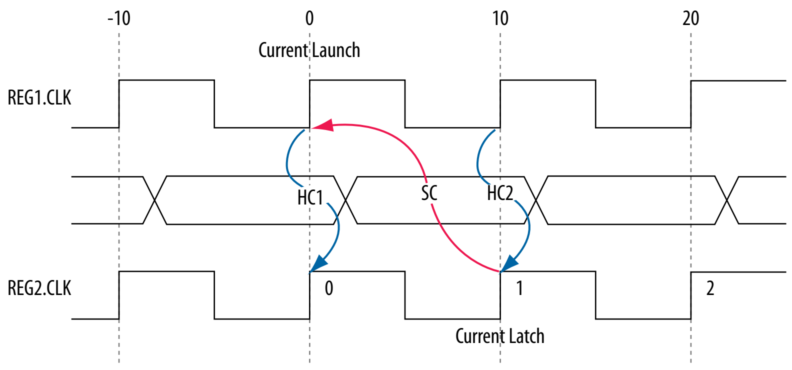
Each example explains how the multicycle exceptions affect the default setup and hold analysis in the Timing Analyzer. The multicycle exceptions are applied to a simple register-to-register circuit. Both the source and destination clocks are set to 10 ns.

8.3.5.7.1 Default Settings

By default, the Timing Analyzer performs a single-cycle analysis to determine the setup and hold checks. Also, by default, the Timing Analyzer sets the end multicycle setup assignment value to one and the end multicycle hold assignment value to zero.

The source and the destination timing waveform for the source register and destination register, respectively where HC1 and HC2 are hold checks one and two and SC is the setup check.

Figure 58. Default Timing Diagram



The calculation that the Timing Analyzer performs to determine the setup check.

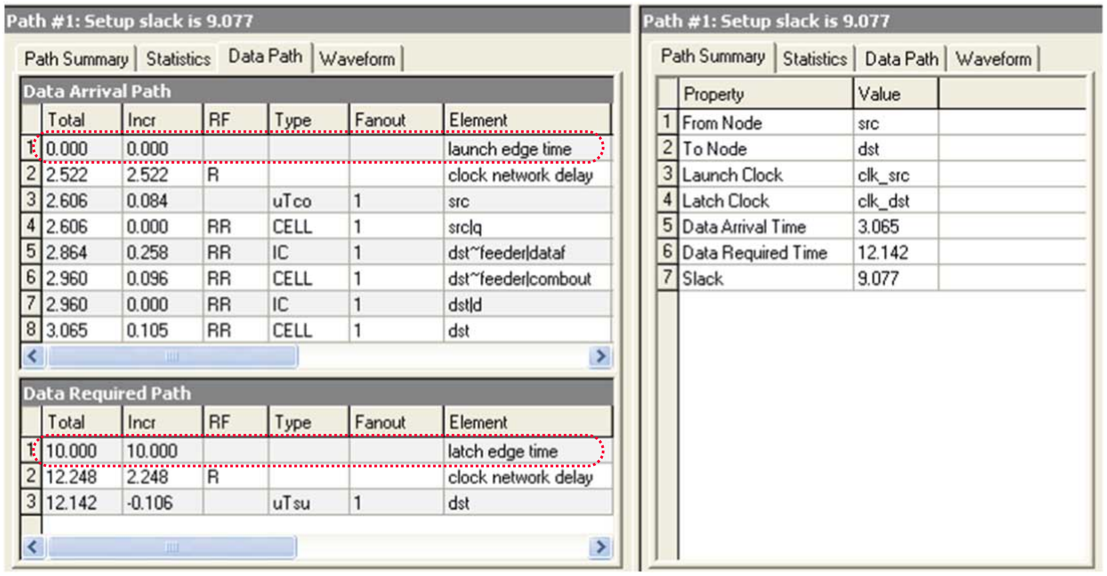
Figure 59. Setup Check

$$\begin{aligned}\text{setup check} &= \text{current latch edge} - \text{closest previous launch edge}\\ &= 10\text{ ns} - 0\text{ ns}\\ &= 10\text{ ns}\end{aligned}$$

The most restrictive setup relationship with the default single-cycle analysis, that is, a setup relationship with an end multicycle setup assignment of one, is 10 ns.

The setup report for the default setup in the Timing Analyzer with the launch and latch edges highlighted.

Figure 60. Setup Report



Path #1: Setup slack is 9.077

Path Summary | Statistics | Data Path | Waveform

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Element
1	0.000	0.000				launch edge time
2	2.522	2.522	R			clock network delay
3	2.606	0.084		uTco	1	src
4	2.606	0.000	RR	CELL	1	src|q
5	2.864	0.258	RR	IC	1	dst~feeder|dataf
6	2.960	0.096	RR	CELL	1	dst~feeder|combout
7	2.960	0.000	RR	IC	1	dst|d
8	3.065	0.105	RR	CELL	1	dst

Data Required Path

	Total	Incr	RF	Type	Fanout	Element
1	10.000	10.000				latch edge time
2	12.248	2.248	R			clock network delay
3	12.142	-0.106		uTsu	1	dst

Path #1: Setup slack is 9.077

Path Summary | Statistics | Data Path | Waveform

	Property	Value
1	From Node	src
2	To Node	dst
3	Launch Clock	clk_src
4	Latch Clock	clk_dst
5	Data Arrival Time	3.065
6	Data Required Time	12.142
7	Slack	9.077

The calculation that the Timing Analyzer performs to determine the hold check. Both hold checks are equivalent.

Figure 61. Hold Check

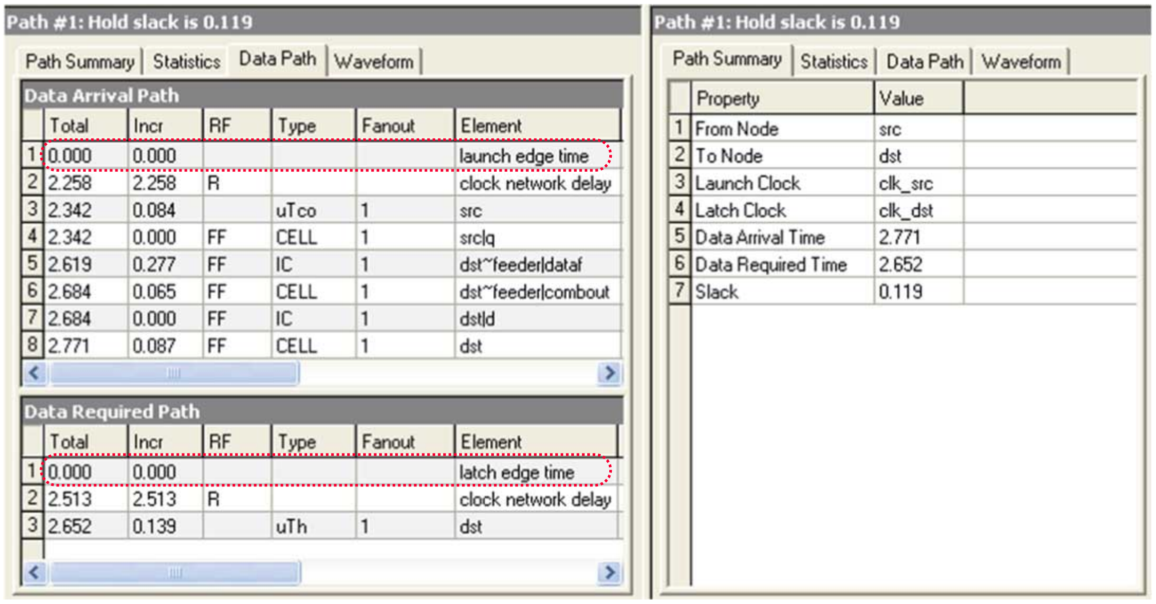
$$\begin{aligned}\text{hold check 1} &= \text{current launch edge} - \text{previous latch edge}\\ &= 0\text{ ns} - 0\text{ ns}\\ &= 0\text{ ns}\end{aligned}$$

$$\begin{aligned}\text{hold check 2} &= \text{next launch edge} - \text{current latch edge}\\ &= 10\text{ ns} - 10\text{ ns}\\ &= 0\text{ ns}\end{aligned}$$

The most restrictive hold relationship with the default single-cycle analysis, that a hold relationship with an end multicycle hold assignment of zero, is 0 ns.

The hold report for the default setup in the Timing Analyzer with the launch and latch edges highlighted.

Figure 62. Hold Report



Path #1: Hold slack is 0.119

Path Summary | Statistics | Data Path | Waveform

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Element
1	0.000	0.000				launch edge time
2	2.258	2.258	R			clock network delay
3	2.342	0.084		uTco	1	src
4	2.342	0.000	FF	CELL	1	src|q
5	2.619	0.277	FF	IC	1	dst~feeder|dataf
6	2.684	0.065	FF	CELL	1	dst~feeder|combout
7	2.684	0.000	FF	IC	1	dst|d
8	2.771	0.087	FF	CELL	1	dst

Data Required Path

	Total	Incr	RF	Type	Fanout	Element
1	0.000	0.000				latch edge time
2	2.513	2.513	R			clock network delay
3	2.652	0.139		uTh	1	dst

Path #1: Hold slack is 0.119

Path Summary | Statistics | Data Path | Waveform

	Property	Value
1	From Node	src
2	To Node	dst
3	Launch Clock	clk_src
4	Latch Clock	clk_dst
5	Data Arrival Time	2.771
6	Data Required Time	2.652
7	Slack	0.119

8.3.5.7.2 End Multicycle Setup = 2 and End Multicycle Hold = 0

In this example, the end multicycle setup assignment value is two, and the end multicycle hold assignment value is zero.

Multicycle Exceptions

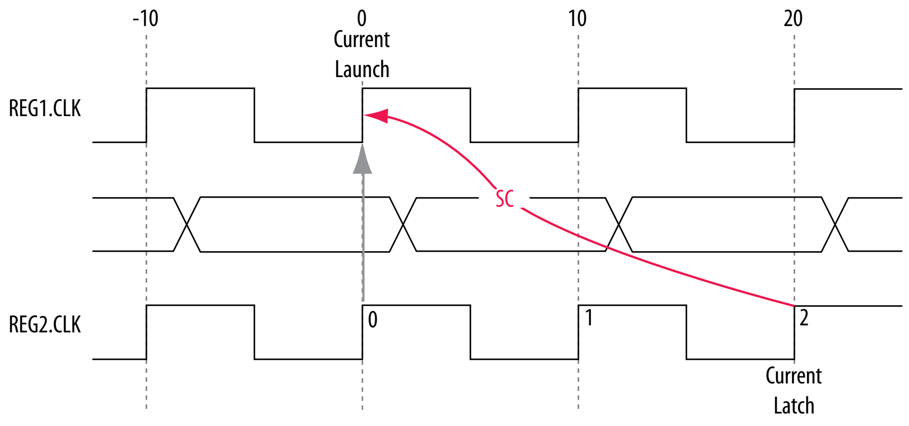
```
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] \
    -setup -end 2
```

Note: An end multicycle hold value is not required because the default end multicycle hold value is zero.

In this example, the setup relationship is relaxed by a full clock period by moving the latch edge to the next latch edge. The hold analysis is unchanged from the default settings.

The setup timing diagram for the analysis that the Timing Analyzer performs. The latch edge is a clock cycle later than in the default single-cycle analysis.

Figure 63. Setup Timing Diagram



The calculation that the Timing Analyzer performs to determine the setup check.

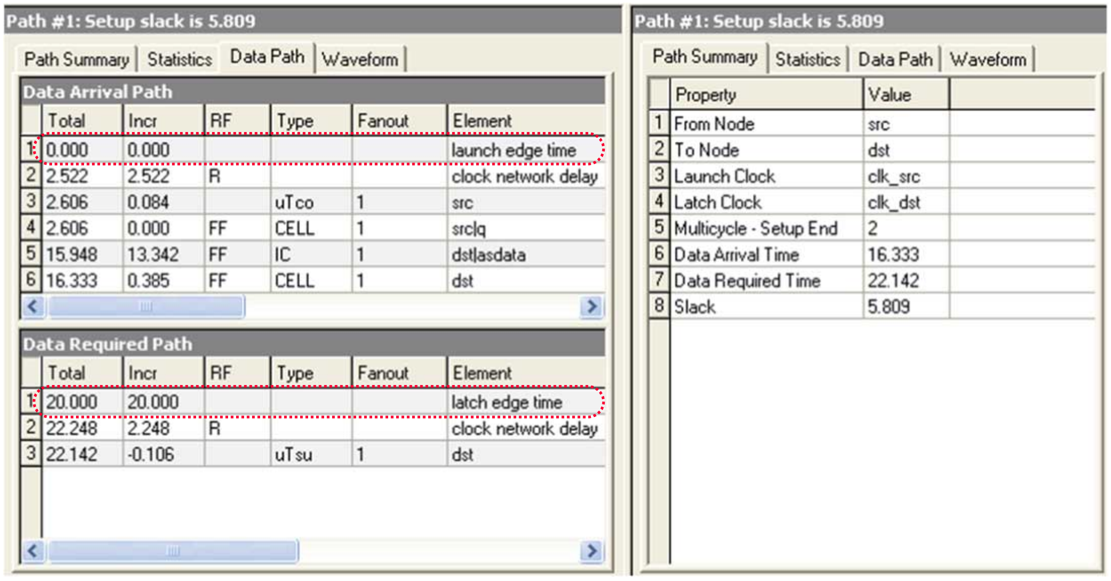
Figure 64. Setup Check

$$\begin{aligned}\text{setup check} &= \text{current latch edge} - \text{closest previous launch edge}\\ &= 20\text{ ns} - 0\text{ ns}\\ &= 20\text{ ns}\end{aligned}$$

The most restrictive setup relationship with an end multicycle setup assignment of two is 20 ns.

The setup report in the Timing Analyzer with the launch and latch edges highlighted.

Figure 65. Setup Report



Path #1: Setup slack is 5.809

Path Summary | Statistics | Data Path | Waveform

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Element
1	0.000	0.000				launch edge time
2	2.522	2.522	R			clock network delay
3	2.606	0.084		uTco	1	src
4	2.606	0.000	FF	CELL	1	src|q
5	15.948	13.342	FF	IC	1	dst|asdata
6	16.333	0.385	FF	CELL	1	dst

Data Required Path

	Total	Incr	RF	Type	Fanout	Element
1	20.000	20.000				latch edge time
2	22.248	2.248	R			clock network delay
3	22.142	-0.106		uTsu	1	dst

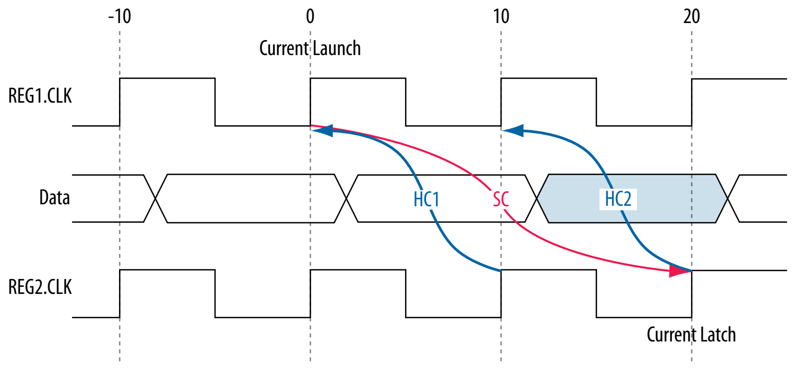
Path #1: Setup slack is 5.809

Path Summary | Statistics | Data Path | Waveform

	Property	Value
1	From Node	src
2	To Node	dst
3	Launch Clock	clk_src
4	Latch Clock	clk_dst
5	Multicycle - Setup End	2
6	Data Arrival Time	16.333
7	Data Required Time	22.142
8	Slack	5.809

Because the multicycle hold latch and launch edges are the same as the results of hold analysis with the default settings, the multicycle hold analysis in this example is equivalent to the single-cycle hold analysis. The hold checks are relative to the setup check. Usually, the Timing Analyzer performs hold checks on every possible setup check, not only on the most restrictive setup check edges.

Figure 66. Hold Timing DIagram



The calculation that the Timing Analyzer performs to determine the hold check. Both hold checks are equivalent.

Figure 67.

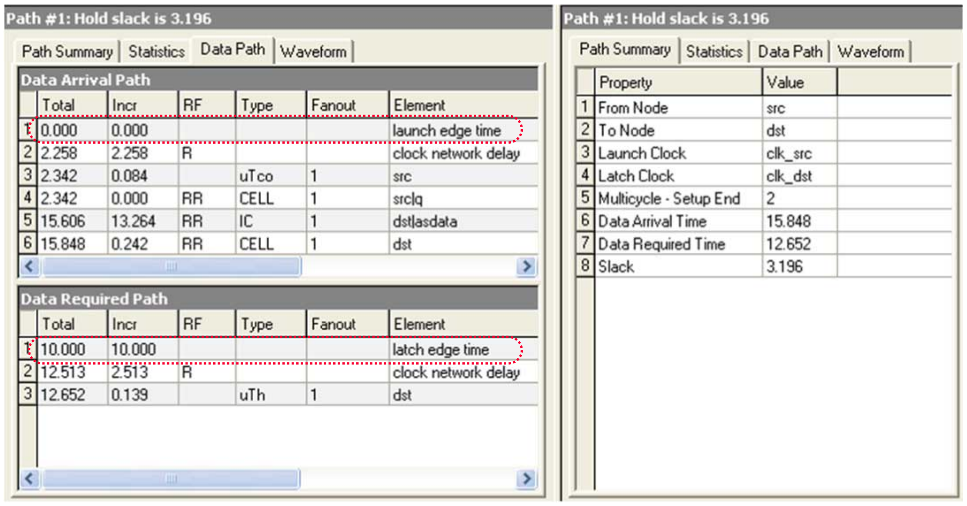
$$\begin{aligned}\text{hold check 1} &= \text{current launch edge} - \text{previous latch edge}\\ &= 0\text{ ns} - 10\text{ ns}\\ &= -10\text{ ns}\end{aligned}$$

$$\begin{aligned}\text{hold check 2} &= \text{next launch edge} - \text{current latch edge}\\ &= 10\text{ ns} - 20\text{ ns}\\ &= -10\text{ ns}\end{aligned}$$

The most restrictive hold relationship with an end multicycle setup assignment value of two and an end multicycle hold assignment value of zero is 10 ns.

The hold report for this example in the Timing Analyzer with the launch and latch edges highlighted.

Figure 68. Hold Report



Path #1: Hold slack is 3.196

Path Summary | Statistics | Data Path | Waveform

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Element
1	0.000	0.000				launch edge time
2	2.258	2.258	R			clock network delay
3	2.342	0.084		uTco	1	src
4	2.342	0.000	RR	CELL	1	src|q
5	15.606	13.264	RR	IC	1	dst|asdata
6	15.848	0.242	RR	CELL	1	dst

Data Required Path

	Total	Incr	RF	Type	Fanout	Element
1	10.000	10.000				latch edge time
2	12.513	2.513	R			clock network delay
3	12.652	0.139		uTh	1	dst

Path #1: Hold slack is 3.196

Path Summary | Statistics | Data Path | Waveform

	Property	Value
1	From Node	src
2	To Node	dst
3	Launch Clock	clk_src
4	Latch Clock	clk_dst
5	Multicycle - Setup End	2
6	Data Arrival Time	15.848
7	Data Required Time	12.652
8	Slack	3.196

8.3.5.7.3 End Multicycle Setup = 2 and End Multicycle Hold = 1

In this example, the end multicycle setup assignment value is two, and the end multicycle hold assignment value is one.

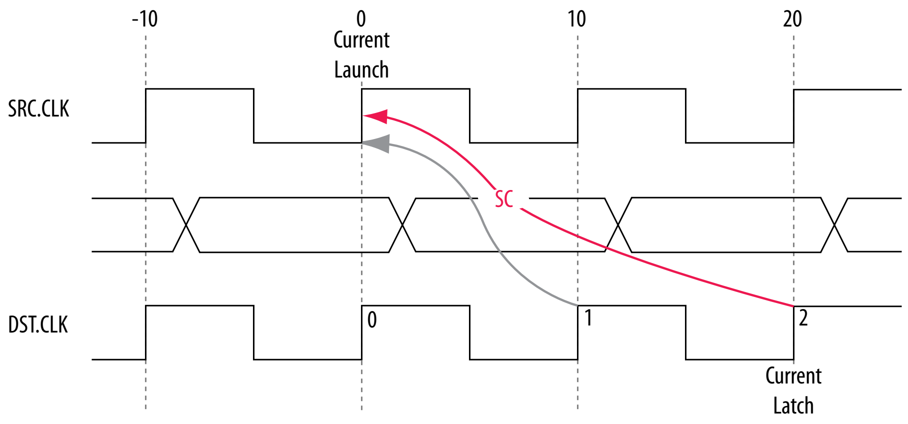
Multicycle Exceptions

```
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] \
    -setup -end 2
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] -hold
-end 1
```

In this example, the setup relationship is relaxed by two clock periods by moving the latch edge to the left two clock periods. The hold relationship is relaxed by a full period by moving the latch edge to the previous latch edge.

The setup timing diagram for the analysis that the Timing Analyzer performs.

Figure 69. Setup Timing Diagram



The calculation that the Timing Analyzer performs to determine the setup check.

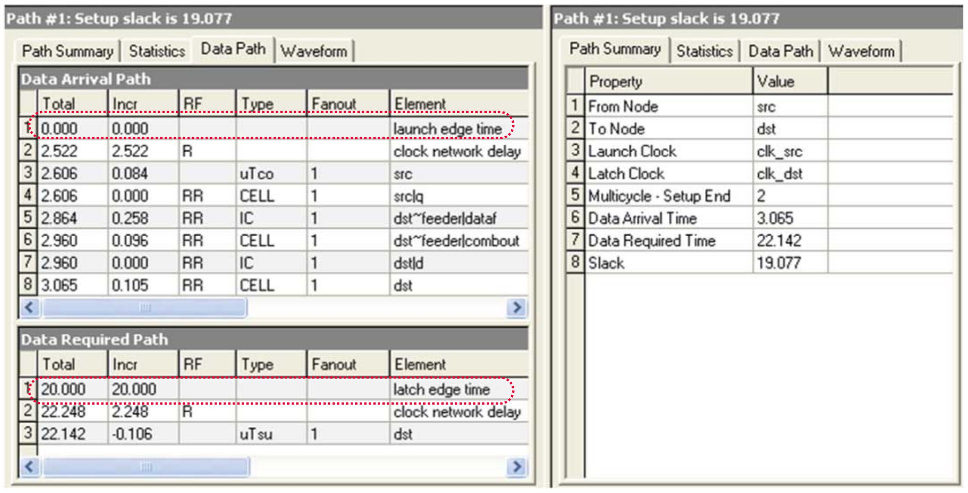
Figure 70. Setup Check

$$\begin{aligned}\text{setup check} &= \text{current latch edge} - \text{closest previous launch edge}\\ &= 20\text{ ns} - 0\text{ ns}\\ &= 20\text{ ns}\end{aligned}$$

The most restrictive hold relationship with an end multicycle setup assignment value of two is 20 ns.

The setup report for this example in the Timing Analyzer with the launch and latch edges highlighted.

Figure 71. Setup Report



Path #1: Setup slack is 19.077

Path Summary | Statistics | Data Path | Waveform

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Element
1	0.000	0.000				launch edge time
2	2.522	2.522	R			clock network delay
3	2.606	0.084		uTco	1	src
4	2.606	0.000	RR	CELL	1	src|q
5	2.864	0.258	RR	IC	1	dst~feeder|dataf
6	2.960	0.096	RR	CELL	1	dst~feeder|combout
7	2.960	0.000	RR	IC	1	dst|d
8	3.065	0.105	RR	CELL	1	dst

Data Required Path

	Total	Incr	RF	Type	Fanout	Element
1	20.000	20.000				latch edge time
2	22.248	2.248	R			clock network delay
3	22.142	-0.106		uTsu	1	dst

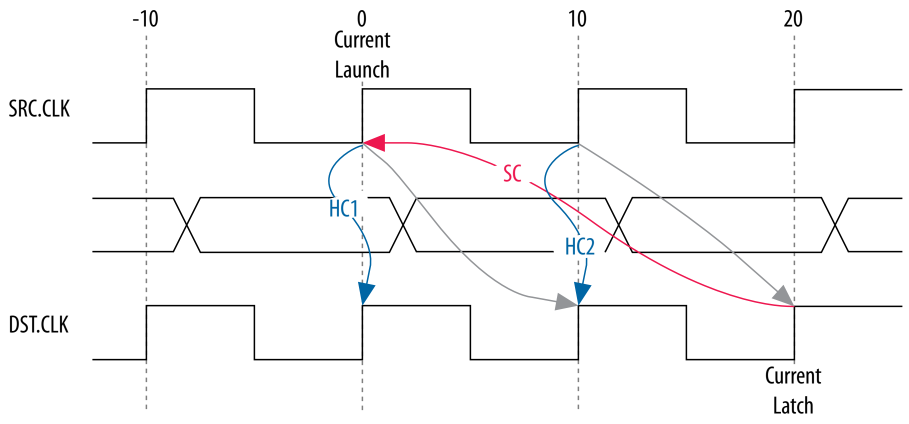
Path #1: Setup slack is 19.077

Path Summary | Statistics | Data Path | Waveform

	Property	Value
1	From Node	src
2	To Node	dst
3	Launch Clock	clk_src
4	Latch Clock	clk_dst
5	Multicycle - Setup End	2
6	Data Arrival Time	3.065
7	Data Required Time	22.142
8	Slack	19.077

The timing diagram for the hold checks for this example. The hold checks are relative to the setup check.

Figure 72. Hold Timing Diagram



The calculation that the Timing Analyzer performs to determine the hold check. Both hold checks are equivalent.

Figure 73. Hold Check

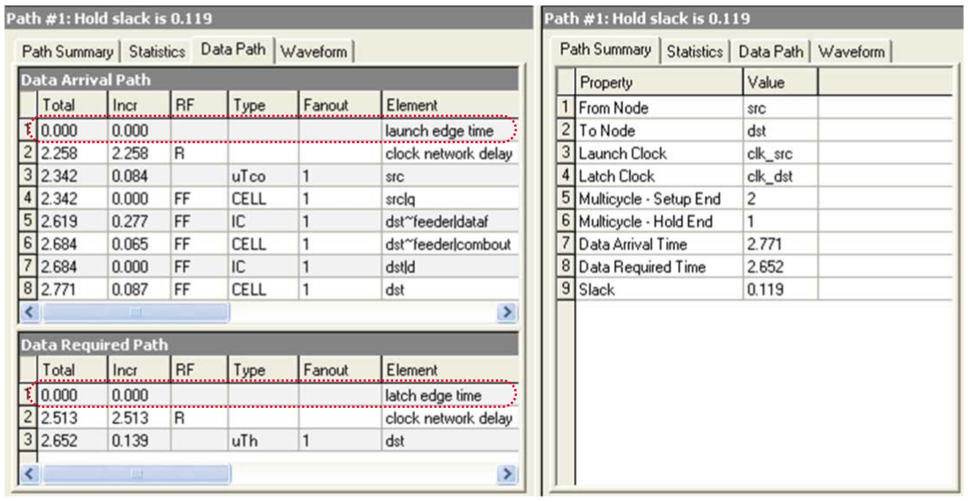
hold check 1 = current launch edge – previous latch edge
= 0 ns – 0 ns
= 0 ns

hold check 2 = next launch edge – current latch edge
= 10 ns – 10 ns
= 0 ns

The most restrictive hold relationship with an end multicycle setup assignment value of two and an end multicycle hold assignment value of one is 0 ns.

The hold report for this example in the Timing Analyzer with the launch and latch edges highlighted.

Figure 74. Hold Report



Path #1: Hold slack is 0.119

Path Summary | Statistics | Data Path | Waveform

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Element
1	0.000	0.000				launch edge time
2	2.258	2.258	R			clock network delay
3	2.342	0.084		uTco	1	src
4	2.342	0.000	FF	CELL	1	src|q
5	2.619	0.277	FF	IC	1	dst~feeder|dataf
6	2.684	0.065	FF	CELL	1	dst~feeder|combout
7	2.684	0.000	FF	IC	1	dst|d
8	2.771	0.087	FF	CELL	1	dst

Data Required Path

	Total	Incr	RF	Type	Fanout	Element
1	0.000	0.000				latch edge time
2	2.513	2.513	R			clock network delay
3	2.652	0.139		uTh	1	dst

Path #1: Hold slack is 0.119

Path Summary | Statistics | Data Path | Waveform

	Property	Value
1	From Node	src
2	To Node	dst
3	Launch Clock	clk_src
4	Latch Clock	clk_dst
5	Multicycle - Setup End	2
6	Multicycle - Hold End	1
7	Data Arrival Time	2.771
8	Data Required Time	2.652
9	Slack	0.119

8.3.5.8 Application of Multicycle Exceptions

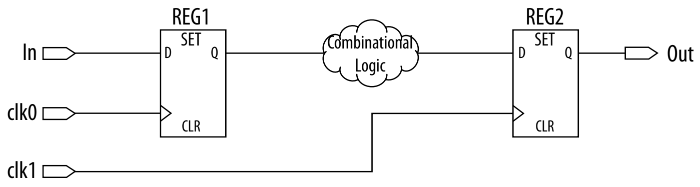
This section shows the following examples of applications of multicycle exceptions. Each example explains how the multicycle exceptions affect the default setup and hold analysis in the Timing Analyzer. All of the examples are between related clock domains. If your design contains related clocks, such as PLL clocks, and paths between related clock domains, you can apply multicycle constraints.

8.3.5.8.1 Same Frequency Clocks with Destination Clock Offset

In this example, the source and destination clocks have the same frequency, but the destination clock is offset with a positive phase shift. Both the source and destination clocks have a period of 10 ns. The destination clock has a positive phase shift of 2 ns with respect to the source clock.

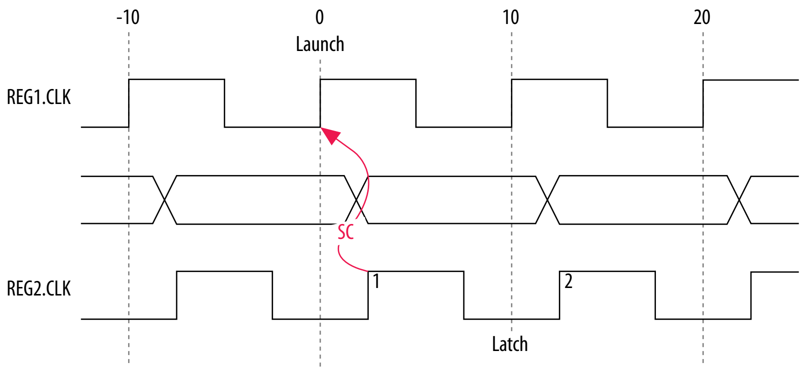
An example of a design with same frequency clocks and a destination clock offset.

Figure 75. Same Frequency Clocks with Destination Clock Offset



The timing diagram for default setup check analysis that the Timing Analyzer performs.

Figure 76. Setup Timing Diagram



The calculation that the Timing Analyzer performs to determine the setup check.

Figure 77. Setup Check

$$\begin{aligned}\text{setup check} &= \text{current latch edge} - \text{closest previous launch edge}\\ &= 2\text{ ns} - 0\text{ ns}\\ &= 2\text{ ns}\end{aligned}$$

The setup relationship shown is too pessimistic and is not the setup relationship required for typical designs. To correct the default analysis, you must use an end multicycle setup exception of two. A multicycle exception used to correct the default analysis in this example in your SDC file.

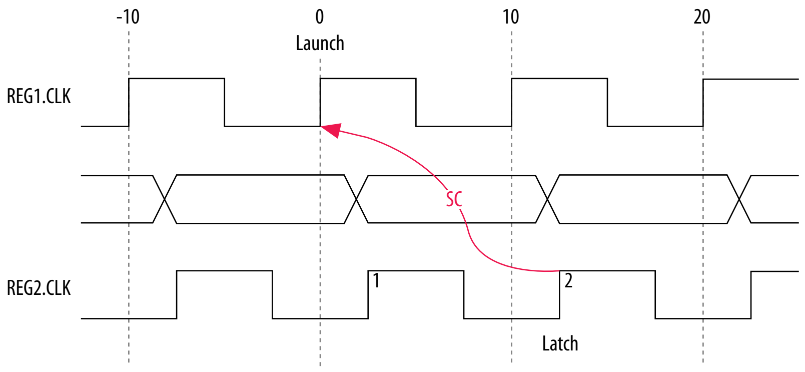
Multicycle Exceptions

```
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] \
     -setup -end 2
```

The timing diagram for the preferred setup relationship for this example.

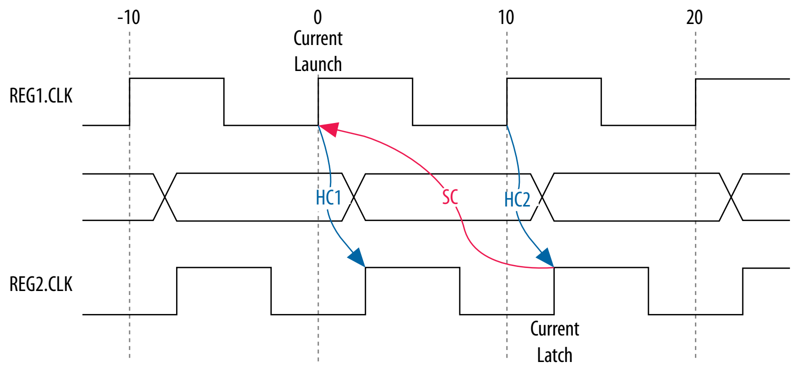
.

Figure 78. Preferred Setup Relationship



The timing diagram for default hold check analysis that the Timing Analyzer performs with an end multicycle setup value of two.

Figure 79. Default Hold Check



The calculation that the Timing Analyzer performs to determine the hold check.

Figure 80. Hold Check

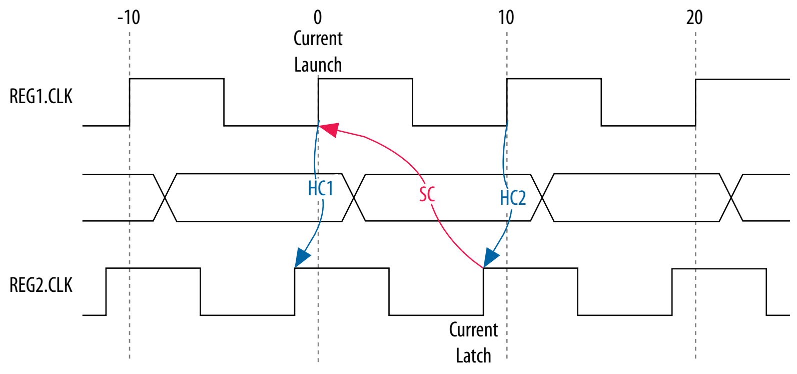
$$\begin{aligned}\text{hold check 1} &= \text{current launch edge} - \text{previous latch edge}\\ &= 0\text{ ns} - 2\text{ ns}\\ &= -2\text{ ns}\end{aligned}$$

$$\begin{aligned}\text{hold check 2} &= \text{next launch edge} - \text{current latch edge}\\ &= 10\text{ ns} - 12\text{ ns}\\ &= -2\text{ ns}\end{aligned}$$

In this example, the default hold analysis returns the preferred hold requirements and no multicycle hold exceptions are required.

The associated setup and hold analysis if the phase shift is –2 ns. In this example, the default hold analysis is correct for the negative phase shift of 2 ns, and no multicycle exceptions are required.

Figure 81. Negative Phase Shift

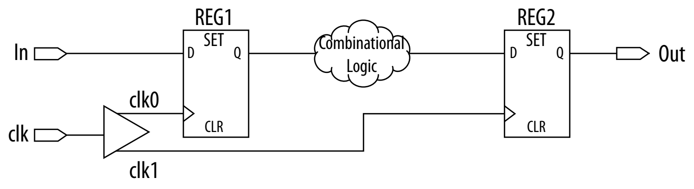


8.3.5.8.2 Destination Clock Frequency is a Multiple of the Source Clock Frequency

In this example, the destination clock frequency value of 5 ns is an integer multiple of the source clock frequency of 10 ns. The destination clock frequency can be an integer multiple of the source clock frequency when a PLL is used to generate both clocks with a phase shift applied to the destination clock.

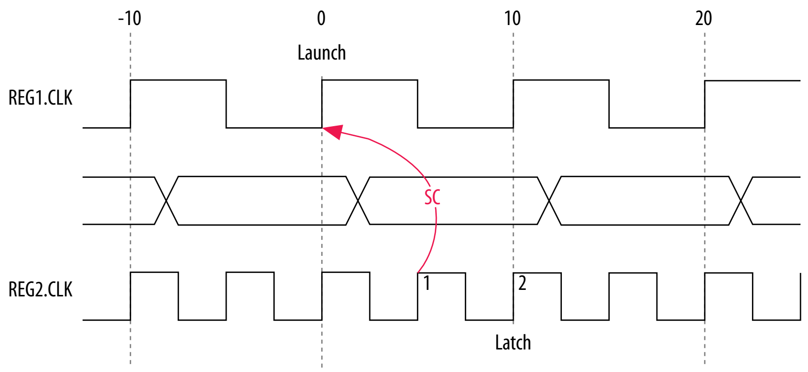
An example of a design where the destination clock frequency is a multiple of the source clock frequency.

Figure 82. Destination Clock is Multiple of Source Clock



The timing diagram for default setup check analysis that the Timing Analyzer performs.

Figure 83. Setup Timing Diagram



The calculation that the Timing Analyzer performs to determine the setup check.

Figure 84. Setup Check

$$
\begin{aligned}
\text{setup check} &= \text{current latch edge} - \text{closest previous launch edge} \\
&= 5\text{ ns} - 0\text{ ns} \\
&= 5\text{ ns}
\end{aligned}
$$

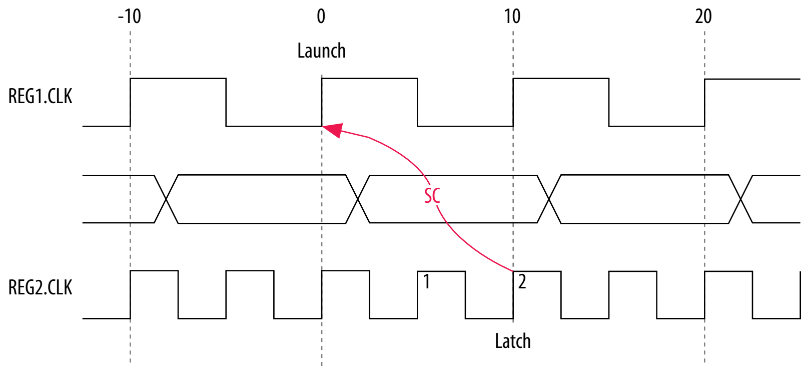
The setup relationship demonstrates that the data does not need to be captured at edge one, but can be captured at edge two; therefore, you can relax the setup requirement. To correct the default analysis, you must shift the latch edge by one clock period with an end multicycle setup exception of two. The multicycle exception assignment used to correct the default analysis in this example.

Multicycle Exceptions

```
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] \
     -setup -end 2
```

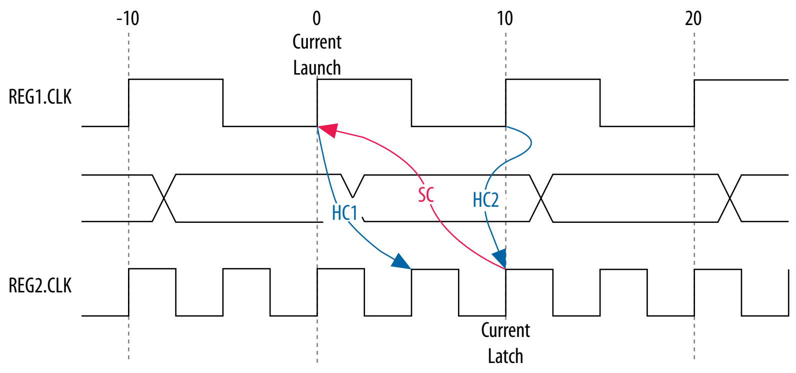
The timing diagram for the preferred setup relationship for this example.

Figure 85. Preferred Setup Analysis



The timing diagram for default hold check analysis performed by the Timing Analyzer with an end multicycle setup value of two.

Figure 86. Default Hold Check



The calculation that the Timing Analyzer performs to determine the hold check.

Figure 87. Hold Check

$$\begin{aligned}\text{hold check 1} &= \text{current launch edge} - \text{previous latch edge}\\ &= 0\text{ ns} - 5\text{ ns}\\ &= -5\text{ ns}\end{aligned}$$

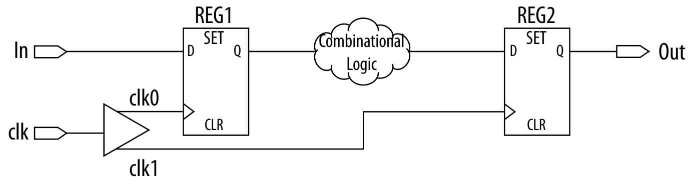
$$\begin{aligned}\text{hold check 2} &= \text{next launch edge} - \text{current latch edge}\\ &= 10\text{ ns} - 10\text{ ns}\\ &= 0\text{ ns}\end{aligned}$$

In this example, hold check one is too restrictive. The data is launched by the edge at 0 ns and should check against the data captured by the previous latch edge at 0 ns, which does not occur in hold check one. To correct the default analysis, you must use an end multicycle hold exception of one.

8.3.5.8.3 Destination Clock Frequency is a Multiple of the Source Clock Frequency with an Offset

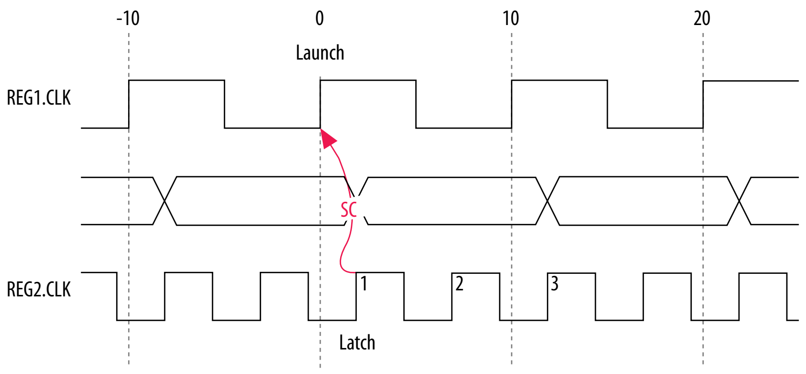
This example is a combination of the previous two examples. The destination clock frequency is an integer multiple of the source clock frequency and the destination clock has a positive phase shift. The destination clock frequency is 5 ns and the source clock frequency is 10 ns. The destination clock also has a positive offset of 2 ns with respect to the source clock. The destination clock frequency can be an integer multiple of the source clock frequency with an offset when a PLL is used to generate both clocks with a phase shift applied to the destination clock. The following example shows a design in which the destination clock frequency is a multiple of the source clock frequency with an offset.

Figure 88. Destination Clock is Multiple of Source Clock with Offset



The timing diagram for the default setup check analysis the Timing Analyzer performs.

Figure 89. Setup Timing Diagram



The calculation that the Timing Analyzer performs to determine the setup check.

Figure 90. Hold Check

$$\begin{aligned}\text{setup check} &= \text{current latch edge} - \text{closest previous launch edge}\\ &= 2\text{ ns} - 0\text{ ns}\\ &= 2\text{ ns}\end{aligned}$$

The setup relationship in this example demonstrates that the data does not need to be captured at edge one, but can be captured at edge two; therefore, you can relax the setup requirement. To correct the default analysis, you must shift the latch edge by one clock period with an end multicycle setup exception of three.

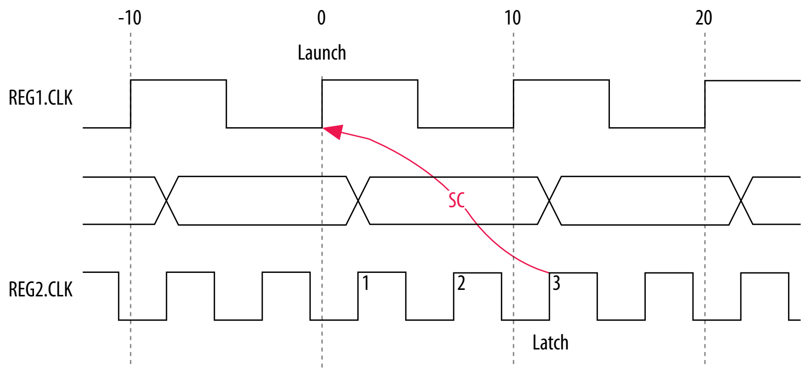
The multicycle exception code you can use to correct the default analysis in this example.

Multicycle Exceptions

```
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] \
     -setup -end 3
```

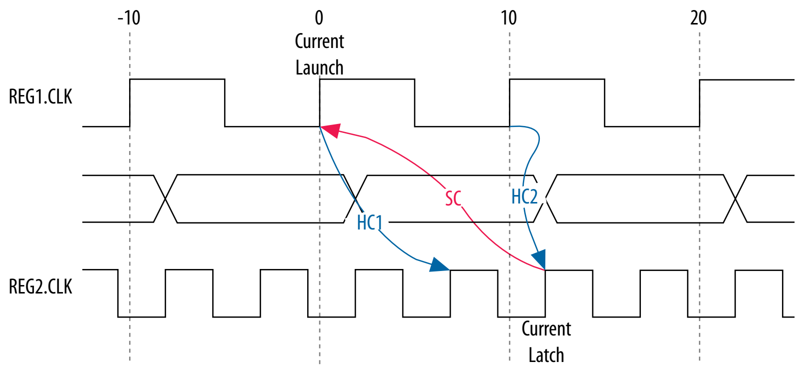
The timing diagram for the preferred setup relationship for this example.

Figure 91. Preferred Setup Analysis



The timing diagram for default hold check analysis the Timing Analyzer performs with an end multicycle setup value of three.

Figure 92. Default Hold Check



The calculation that the Timing Analyzer performs to determine the hold check.

Figure 93. Hold Check

$$\begin{aligned}\text{hold check 1} &= \text{current launch edge} - \text{previous latch edge}\\ &= 0\text{ ns} - 5\text{ ns}\\ &= -5\text{ ns}\end{aligned}$$

$$\begin{aligned}\text{hold check 2} &= \text{next launch edge} - \text{current latch edge}\\ &= 10\text{ ns} - 10\text{ ns}\\ &= 0\text{ ns}\end{aligned}$$

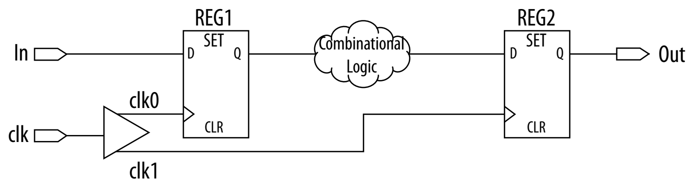
In this example, hold check one is too restrictive. The data is launched by the edge at 0 ns and should check against the data captured by the previous latch edge at 2 ns, which does not occur in hold check one. To correct the default analysis, you must use an end multicycle hold exception of one.

8.3.5.8.4 Source Clock Frequency is a Multiple of the Destination Clock Frequency

In this example, the source clock frequency value of 5 ns is an integer multiple of the destination clock frequency of 10 ns. The source clock frequency can be an integer multiple of the destination clock frequency when a PLL is used to generate both clocks and different multiplication and division factors are used.

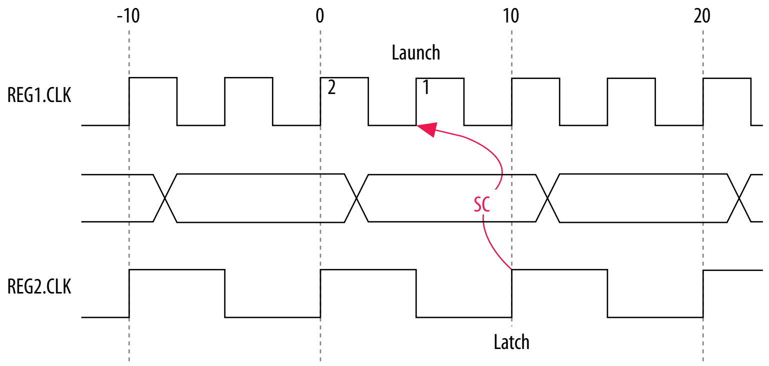
An example of a design where the source clock frequency is a multiple of the destination clock frequency.

Figure 94. Source Clock Frequency is Multiple of Destination Clock Frequency



The timing diagram for default setup check analysis performed by the Timing Analyzer.

Figure 95. Default Setup Check Analysis



The calculation that the Timing Analyzer performs to determine the setup check.

Figure 96. Setup Check

$$\begin{aligned}\text{setup check} &= \text{current latch edge} - \text{closest previous launch edge}\\ &= 10\text{ ns} - 5\text{ ns}\\ &= 5\text{ ns}\end{aligned}$$

The setup relationship shown demonstrates that the data launched at edge one does not need to be captured, and the data launched at edge two must be captured; therefore, you can relax the setup requirement. To correct the default analysis, you must shift the launch edge by one clock period with a start multicycle setup exception of two.

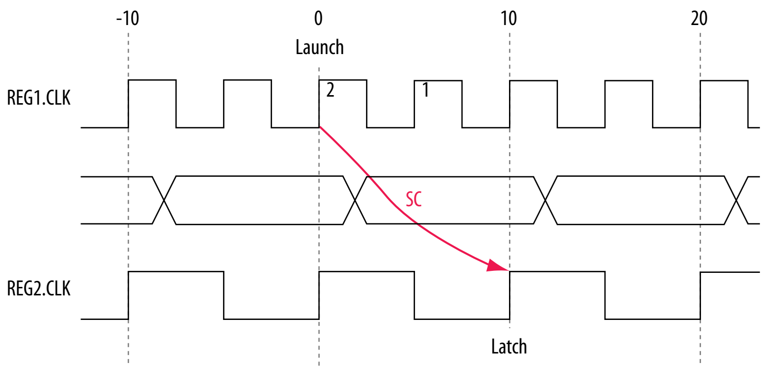
The multicycle exception code you can use to correct the default analysis in this example.

Multicycle Exceptions

```
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] \
     -setup -start 2
```

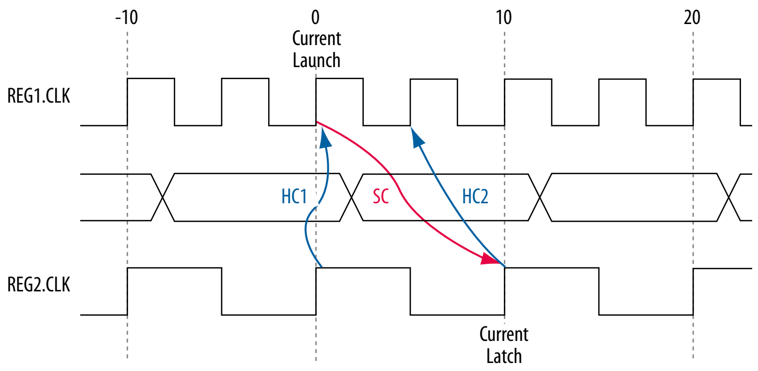
The timing diagram for the preferred setup relationship for this example.

Figure 97. Preferred Setup Check Analysis



The timing diagram for default hold check analysis the Timing Analyzer performs for a start multicycle setup value of two.

Figure 98. Default Hold Check



The calculation that the Timing Analyzer performs to determine the hold check.

Figure 99. Hold Check

$$\begin{aligned}\text{hold check 1} &= \text{current launch edge} - \text{previous latch edge}\\ &= 0\text{ ns} - 0\text{ ns}\\ &= 0\text{ ns}\end{aligned}$$

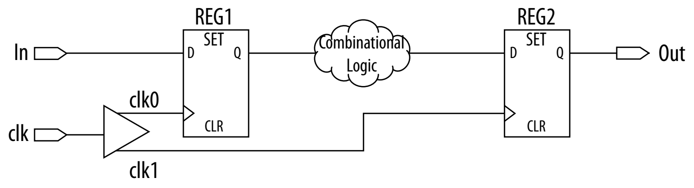
$$\begin{aligned}\text{hold check 2} &= \text{next launch edge} - \text{current latch edge}\\ &= 5\text{ ns} - 10\text{ ns}\\ &= -5\text{ ns}\end{aligned}$$

In this example, hold check two is too restrictive. The data is launched next by the edge at 10 ns and should check against the data captured by the current latch edge at 10 ns, which does not occur in hold check two. To correct the default analysis, you must use a start multicycle hold exception of one.

8.3.5.8.5 Source Clock Frequency is a Multiple of the Destination Clock Frequency with an Offset

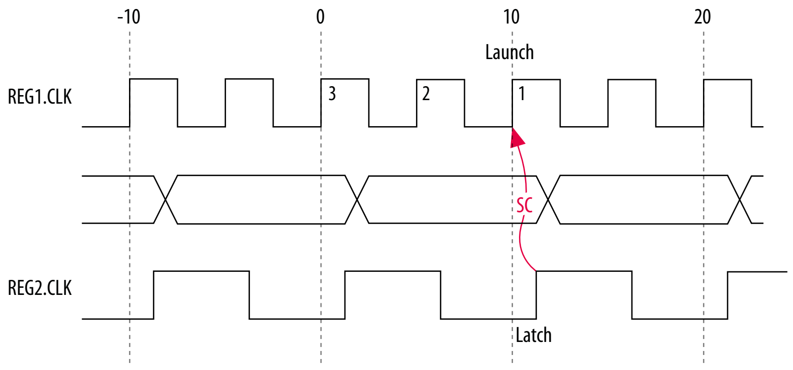
In this example, the source clock frequency is an integer multiple of the destination clock frequency and the destination clock has a positive phase offset. The source clock frequency is 5 ns and destination clock frequency is 10 ns. The destination clock also has a positive offset of 2 ns with respect to the source clock. The source clock frequency can be an integer multiple of the destination clock frequency with an offset when a PLL is used to generate both clocks, different multiplication.

Figure 100. Source Clock Frequency is Multiple of Destination Clock Frequency with Offset



Timing diagram for default setup check analysis the Timing Analyzer performs.

Figure 101. Setup Timing Diagram



The calculation that the Timing Analyzer performs to determine the setup check.

Figure 102. Setup Check

$$\begin{aligned} \text{setup check} &= \text{current latch edge} - \text{closest previous launch edge} \\ &= 12\text{ ns} - 10\text{ ns} \\ &= 2\text{ ns} \end{aligned}$$

The setup relationship in this example demonstrates that the data is not launched at edge one, and the data that is launched at edge three must be captured; therefore, you can relax the setup requirement. To correct the default analysis, you must shift the launch edge by two clock periods with a start multicycle setup exception of three.

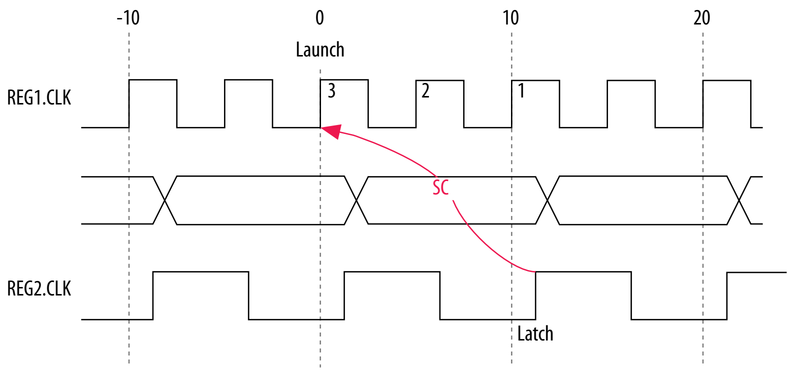
The multicycle exception used to correct the default analysis in this example.

Multicycle Exceptions

```
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] \
     -setup -start 3
```

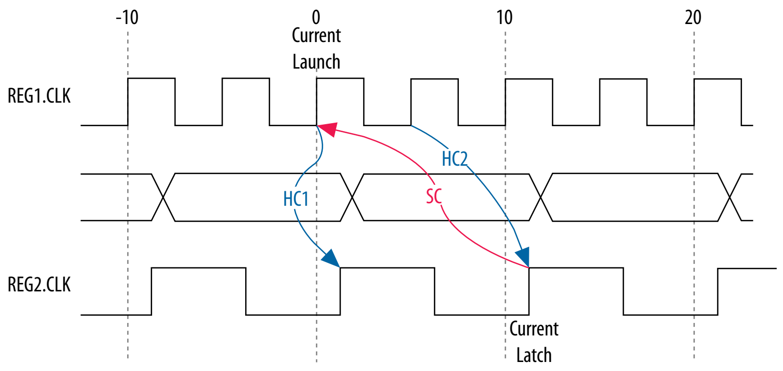
The timing diagram for the preferred setup relationship for this example.

Figure 103. Preferred Setup Check Analysis



The timing diagram for default hold check analysis the Timing Analyzer performs for a start multicycle setup value of three.

Figure 104. Default Hold Check Analysis



The calculation that the Timing Analyzer performs to determine the hold check.

Figure 105. Hold Check

$$\begin{aligned}\text{hold check 1} &= \text{current launch edge} - \text{previous latch edge}\\ &= 0\text{ ns} - 2\text{ ns}\\ &= -2\text{ ns}\end{aligned}$$

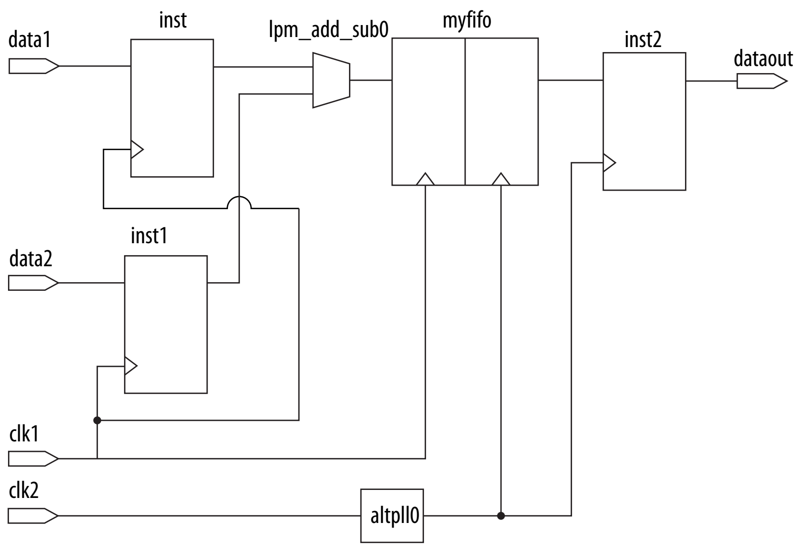
$$\begin{aligned}\text{hold check 2} &= \text{next launch edge} - \text{current latch edge}\\ &= 5\text{ ns} - 12\text{ ns}\\ &= -7\text{ ns}\end{aligned}$$

In this example, hold check two is too restrictive. The data is launched next by the edge at 10 ns and should check against the data captured by the current latch edge at 12 ns, which does not occur in hold check two. To correct the default analysis, you must use a start multicycle hold exception of one.

8.3.6 A Sample Design with SDC File

An example circuit that includes two clocks, a phase-locked loop (PLL), and other common synchronous design elements helps demonstrate how to constrain your design with an SDC file.

Figure 106. Timing Analyzer Constraint Example



The following SDC file contains basic constraints for the example circuit.

Example 12. Basic SDC Constraints

```
# Create clock constraints
create_clock -name clockone -period 10.000 [get_ports {clk1}]
create_clock -name clocktwo -period 10.000 [get_ports {clk2}]
# Create virtual clocks for input and output delay constraints
create clock -name clockone_ext -period 10.000
create clock -name clocktwo_ext -period 10.000
derive_pll_clocks
# derive clock uncertainty
```

```
derive_clock_uncertainty
# Specify that clockone and clocktwo are unrelated by assinging
# them to seperate asynchronus groups
set_clock_groups \
  -asynchronous \
  -group {clockone} \
  -group {clocktwo altpll0|altpll_component|auto_generated|pll1|clk[0]}

# set input and output delays
set_input_delay -clock { clockone_ext } -max 4 [get_ports
{data1}]set_input_delay -clock { clockone_ext } -min -1 [get_ports {data1}]
set_input_delay -clock { clockone_ext } -max 4 [get_ports
{data2}]set_input_delay -clock { clockone_ext } -min -1 [get_ports {data2}]
set_output_delay -clock { clocktwo_ext } -max 6 [get_ports {dataout}]
set_output_delay -clock { clocktwo_ext } -min -3 [get_ports {dataout}]
```

The SDC file contains the following basic constraints you should include for most designs:

- Definitions of `clockone` and `clocktwo` as base clocks, and assignment of those settings to nodes in the design.
- Definitions of `clockone_ext` and `clocktwo_ext` as virtual clocks, which represent clocks driving external devices interfacing with the FPGA.
- Automated derivation of generated clocks on PLL outputs.
- Derivation of clock uncertainty.
- Specification of two clock groups, the first containing `clockone` and its related clocks, the second containing `clocktwo` and its related clocks, and the third group containing the output of the PLL. This specification overrides the default analysis of all clocks in the design as related to each other.
- Specification of input and output delays for the design.

Related Links

Asynchronous Clock Groups on page 99
For more information about asynchronous clock groups.

8.4 Running the Timing Analyzer

When you compile a design, the Timing Analyzer automatically performs multi-corner signoff timing analysis after the Fitter has finished.

- To open the Timing Analyzer GUI directly from the Intel Quartus Prime software GUI, click **Timing Analyzer** on the Tools menu.
- To peform or repeat multi-corner timing analysis from the Intel Quartus Prime GUI, click **Processing ➤ Start ➤ Start Timing Analyzer**.
- To perform multi-corner timing analysis from a system command prompt, type `quartus_sta <`*`options`*`><`*`project_name`*`>`.
- To run the Timing Analyzer as a stand-alone GUI application, type the following command at the command prompt:`quartus_staw`.
- To run the Timing Analyzer in interactive command-shell mode, type the following command at a system command prompt: `quartus_sta -s <`*`options`*`><`*`project_name`*`>`.

The following table lists the command-line options available for the `quartus_sta` executable.

Table 32. Summary of Command-Line Options

Command-Line Option	Description
`-h | --help`	Provides help information on quartus_sta.
`-t <script file> | --script=`*<script file>*	Sources the *<script file>*.
`-s | --shell`	Enters shell mode.
`--tcl_eval <tcl command>`	Evaluates the Tcl command *<tcl command>*.

`--do_report_timing`	For all clocks in the design, run the following commands:
	`report_timing -npaths 1 -to_clock $clock`
	`report_timing -setup -npaths 1 -to_clock $clock`
	`report_timing -hold -npaths 1 -to_clock $clock`
	`report_timing -recovery -npaths 1 -to_clock $clock`
	`report_timing -removal -npaths 1 -to_clock $clock`

`--force_dat`	Forces an update of the project database with new delay information.
`--lower_priority`	Lowers the computing priority of the `quartus_sta` process.
`--post_map`	Uses the post-map database results.
`--sdc=`*<SDC file>*	Specifies the SDC file to use.
`--report_script=<script>`	Specifies a custom report script to call.
`--speed=<value>`	Specifies the device speed grade used for timing analysis.
`--tq2pt`	Generates temporary files to convert the Timing Analyzer SDC file(s) to a PrimeTime SDC file.
`-f <argument file>`	Specifies a file containing additional command-line arguments.
`-c <revision name> | --rev=<revision_name>`	Specifies which revision and its associated Intel Quartus Prime Settings File (`.qsf`) to use.
`--multicorner`	Specifies that all slack summary reports be generated for both slow- and fast-corners.
`--multicorner[=on|off]`	Turns off multicorner timing analysis.
`--voltage=<value_in_mV>`	Specifies the device voltage, in mV used for timing analysis.
`--temperature=<value_in_C>`	Specifies the device temperature in degrees Celsius, used for timing analysis.
`--parallel [=<num_processors>]`	Specifies the number of computer processors to use on a multiprocessor system.
`--64bit`	Enables 64-bit version of the executable.

Related Links

- Constraining and Analyzing with Tcl Commands on page 153
  For more information about using Tcl commands to constrain and analyze your design
- Recommended Flow for First Time Users on page 80
  For more information about steps to perform before opening the Timing Analyzer.

8.4.1 Intel Quartus Prime Settings

Within the Intel Quartus Prime software, there are a number of quick steps for setting up your design with Timing Analyzer. You can modify the appropriate settings in **Assignments ➤ Settings**.

In the **Settings** dialog box, select **Timing Analyzer** in the **Category** list.

The **Timing Analyzer** settings page is where you specify the title and location for a Synopsis Design Constraint (SDC) file. The SDC file is an inudstry standard format for specifying timing constraints. If no SDC file exists, you can create one based on the instructions in this document. The Intel Quartus Prime software provides an SDC template you can use to create your own.

The following Timing Analyzer options should be on by default:

- Enable multicorner timing analysis—Directs the Timing Analyzer to analyze all the timing models of your FPGA against your constraints. This is required for final timing sign-off. Unchecked, only the slow timing model is be analyzed.
- Enable common clock path pessimism removal— Prevents timing analysis from over-calculating the effects of **On-Die Variation**. This makes timing better, and there really is no reason for this to be disabled.
- Report worst-case paths during compilation—This optional setting displays summary of the worst paths in your timing report. This type of path analysis is covered in more detail later in this document. While useful, this summary can increase the size of the `<project>.sta.rpt` with all of these paths.
- Tcl script file for custom reports—This optional setting should prove useful later, allowing you to add custom reports to create a custom analysis. For example, if you are only working on a portion of the full FPGA, you may want additional timing reports that cover that hierarchy.

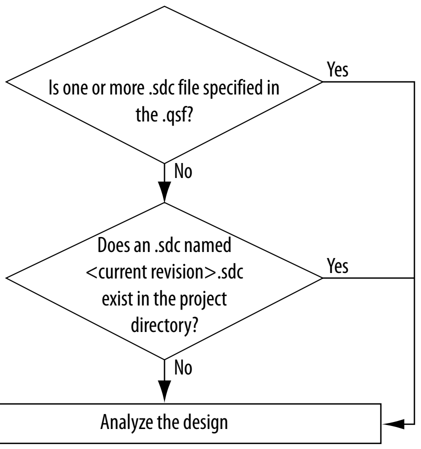
Note: In addition, certain values are set by default. The default duty-cycle is 50% and the default clock frequency is 1Ghz.

8.4.2 SDC File Precedence

The Fitter and the Timing Analyzer process SDC files in the order you specify in the Intel Quartus Prime Settings File (`.qsf`). You can add and remove SDC files to process and specify the order they are processed from the **Assignments** menu.

Click **Settings**, then **Timing Analyzer** and add or remove SDC files, or specify a processing order in the **SDC files to include in the project** box. When you create a new SDC file for a project, you must add it to the project for it to be read during fitting and timing analysis. If you use the Intel Quartus Prime Text Editor to create an SDC file, the option to add it to the project is enabled by default when you save the file. If you use any other editor to create an SDC file, you must remember to add it to the project. If no SDC files are listed in the `.qsf`, the Intel Quartus Prime software looks for an SDC named `<current revision>.sdc` in the project directory. When you use IP from Intel, and some third-parties, the SDC files are often included in a project through an intermediate file called a Intel Quartus Prime IP File (`.qip`). A `.qip` file points to all source files and constraints for a particular IP. If SDC files for IP blocks in your design are included through with a `.qip`, do not re-add them manually. An SDC file can also be added from a Intel Quartus Prime IP File (`.qip`) included in the `.qsf`.

Figure 107. .sdc File Order of Precedence


Is one or more .sdc file specified in the .qsf?
Yes
No
Does an .sdc named <current revision>.sdc exist in the project directory?
Yes
No
Analyze the design

Note: If you type the `read_sdc` command at the command line without any arguments, the Timing Analyzer reads constraints embedded in HDL files, then follows the SDC file precedence order.

The SDC file must contain only SDC commands that specify timing constraints. There are some techniques to control which constraints are applied during different parts of the compilation flow. Tcl commands to manipulate the timing netlist or control the compilation must be in a separate Tcl script.

8.5 Understanding Results

Knowing how your constraints are displayed when analyzing a path is one of the most important skills of timing analysis. This information completes your understanding of timing analysis and lets you correlate the SDC input to the back-end analysis, and determine how the delays in the FPGA affect timing.

8.5.1 Iterative Constraint Modification

Sometimes it is useful to change an SDC constraint and reanalyze the timing results. This flow is particularly common when you are creating timing constraints and want to ensure that constraints are applied appropriately during compilation and timing analysis.

Use the following steps when you iteratvely modify constraints:

1. Open the Timing Analyzer
2. Generate the appropriate reports.
3. Analyze your results
4. Edit your SDC file and save
5. Double-click **Reset Design**
6. Generate the appropriate reports.
7. Analyze your results
8. Repeat steps 4-7 as necessary.

Open the Timing Analyzer—It is most common to use this interactive approach in the Timing Analyzer GUI. You can also use the command-line shell mode, but it does not include some of the time-saving automatic features in the GUI.

Generate the appropriate reports —Use the **Report All Summaries** task under **Macros** to generate setup, hold, recovery, and removal summaries, as well as minimum pulse width checks, and a list of all the defined clocks. These summaries cover all constrained paths in your design. Especially when you are modifying or correcting constraints, you should also perform the Diagnostic task to create reports to identify unconstrained parts of your design, or ignored constraints. Double-click any of the report tasks to automatically run the three tasks under **Netlist Setup** if they haven't already run. One of those tasks reads all SDC files.

Analyze your results—When you are modifying or correcting constraints, review the reports to find any unexpected results. For example, a cross-domain path might indicate that you forgot to cut a transfer by including a clock in a clock group.

Edit your SDC file and save it—Create or edit the appropriate constraints in your SDC files. If you edit your SDC file in the Intel Quartus Prime Tex Editor, you can benefit from tooltips showing constraint options, and dialog boxes that guide you when creating constraints.

Reset the design—Double click **Reset Design** task to remove all constraints from your design. Removing all constraints from your design prepares it to reread the SDC files, including your changes.

Be aware that this method just performs timing analysis using new constraints, but the fit being analyzed has not changed. The place-and-route was performed with the old constraints, but you are analyzing with new constraints, so if something is failing timing against these new constraints,you may need to run place-and-route again.

For example, the Fitter may concentrate on a very long path in your design, trying to close timing. For example, you may realize that a path runs at a lower rate, and so have added `set_multicycle_path` assignments to relax the relationship (open the window when data is valid). When you perform Timing Analyzer analysis iteratively with these new multicycles, new paths replace the old. The new paths may have sub-optimal placement since the Fitter was concentrating on the previous paths when it ran, because they were more critical. The iterative method is recommend for getting your SDC files correct, but you should perform a full compilation to see what the Intel Quartus Prime software can do with those constraints.

Related Links

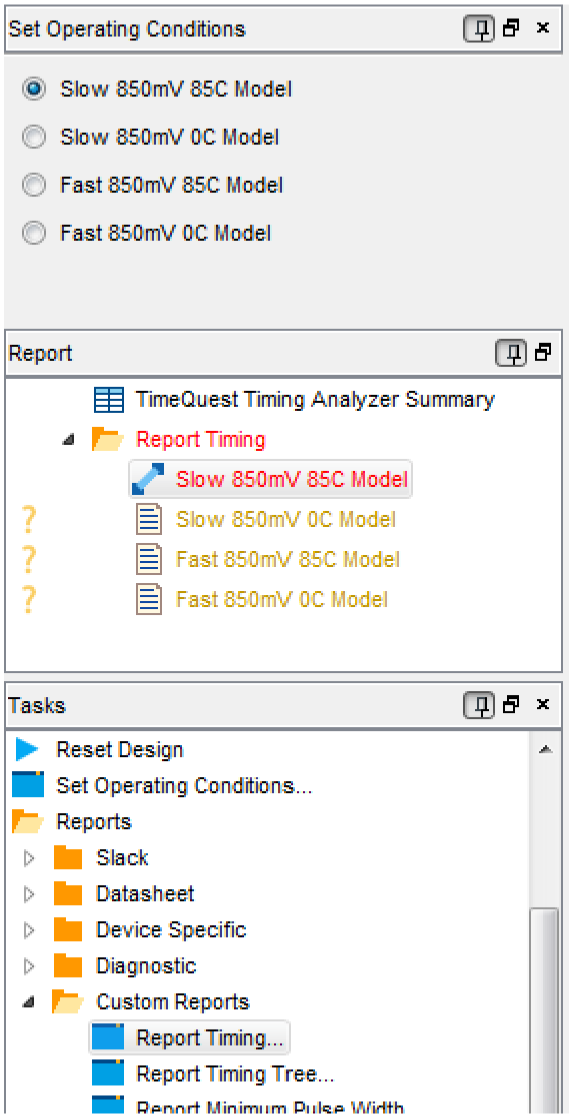
Relaxing Setup with set_multicyle_path on page 111

8.5.2 Set Operating Conditions Dialog Box

You can select different operating conditions to analyze from those used to create the existing timing netlist.

Operating conditions consist of voltage and temperature options that are used together. You can run timing analysis for different delay models without having to delete the existing timing netlist. The Timing Analyzer supports multi-corner timing analysis which you can turn on in the dialog box of the command you are performing. A control has been added to the Timing Analyzer UI where you can select operating conditions and analyze timing for combinations of corners.

Select a voltage/temperature combination and double-click **Report Timing** under **Custom Reports** in the **Tasks** pane.



Reports that fail timing appear in red type, reports that pass appear in black type. Reports that have not yet been run are in gold with a question mark (?). Selecting another voltage/temperature combination creates a new report, but any reports previously run persist.

You can use the following context menu options to generate or regenerate reports in the **Report** window:

- **Regenerate**—Regenerate the selected report.
- **Generate in All Corners**—Generate a timing report using all corners.
- **Regenerate All Out of Date**—Regenerate all reports.
- **Delete All Out of Date**—Flush all the reports that have been run to clear the way for new reports with modifications to timing.

Each operating condition generates its own set of reports which appear in their own folders under the **Reports** list. Reports that have not yet been generated display a '**?**' icon in gold. As each report is generated, the folder is updated with the appropriate output.

Note: Reports for a corner not being generated persist until that particular operating condition is modifyed and a new report is created.

8.5.3 Report Timing (Dialog Box)

Once you are comfortable with the **Report All Summaries** command, the next tool in the Timing Analyzer toolbox is **Report Timing**.

The Timing Analyzer displays reports in the **Report** pane, and is similar to a table of contents for all the reports created. Selecting any name in the **Report** panel displays that report in the main viewing pane.

The main viewing pane shows the Slack for every clock domain. Positive slack is good, saying these paths meet timing by that much. The End Point TNS stands for Total Negative Slack, and is the sum of all slacks for each destination and can be used as a relative assessment of how much a domain is failing.

However, this is just a summary. To get details on any domain, you can right-click that row and select **Report Timing...**.

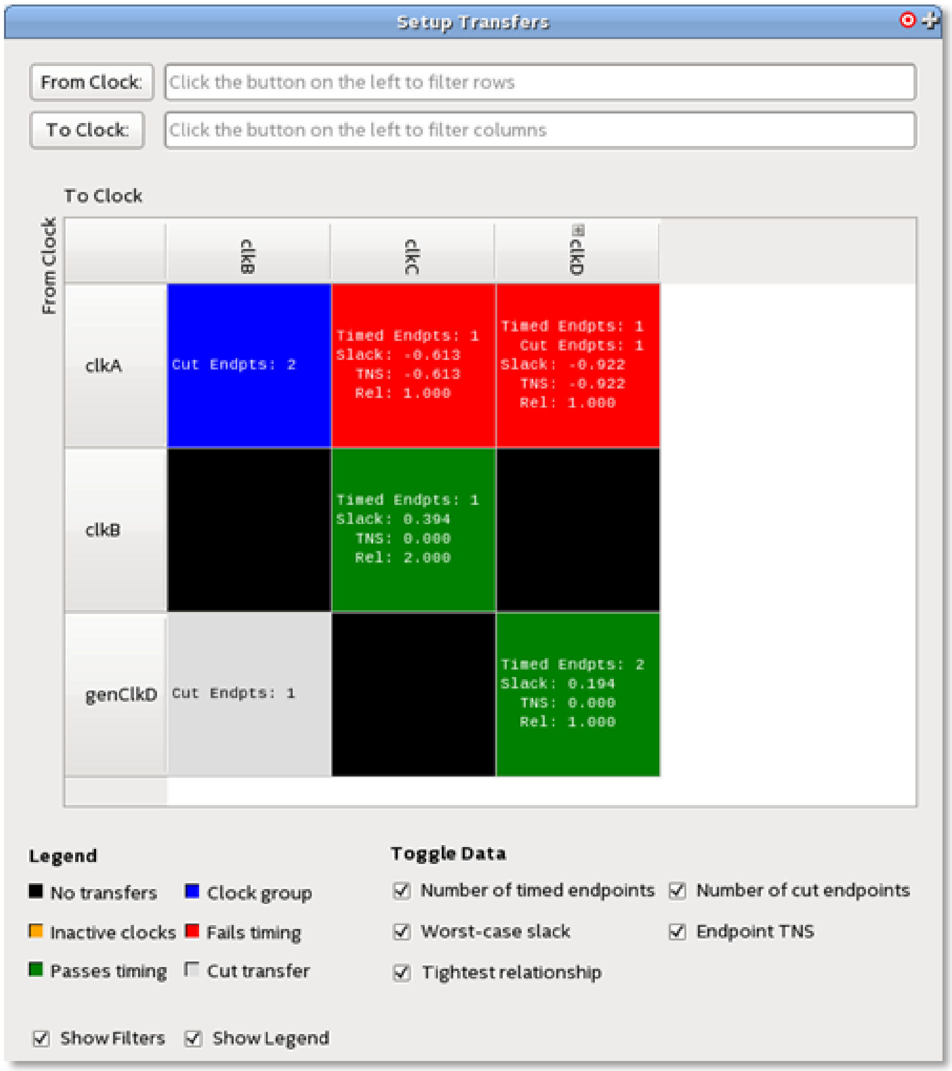
The **Report Timing** dialog box appears, auto-filled with the **Setup** radio button selected and the **To Clock** box filled with the selected clock. This occurs because you were viewing the **Setup Summary** report, and right-clicked on that particular clock. As such, the worst 10 paths where that is the destination clock were reported.You can modify the settingsin various ways, such as increasing the number of paths to report, adding a **Target** filter, adding a **From Clock**, writing the report to a text file, etc.

Note that any `report_timing` command can be copied from the **Console** at the bottom into a user-created Tcl file, so that you can analyze specific paths again in the future without having to negotiate the Timing Analyzer UI. This is often done as users become more comfortable with Timing Analyzer and find themselves analyzing the same problematic parts of their design over and over, but is not required. Many complex designs successfully use Timing Analyzer as a diving tool, i.e. just starting with summaries and diving down into the failing paths after each compile.

8.5.4 Report CDC Viewer Command

The Clock Domain Crossing (CDC) Viewer creates a graphical display which shows either setup, hold, recovery, or removal analysis of all clock transfers in your design. You open this report in the Timing Analyzer GUI by clicking **Reports ➤ Diagnostic ➤ Report CDC Viewer** in the Timing Analyzer. This creates a folder of **CDC Viewer** reports, containing one report for each analysis type.

Figure 108. Setup Transfers Report



Setup Transfers

From Clock: Click the button on the left to filter rows

To Clock: Click the button on the left to filter columns

To Clock

From Clock	clkB	clkC	clkD
clkA	Cut Endpts: 2	Timed Endpts: 1 Slack: -0.613 TNS: -0.613 Rel: 1.000	Timed Endpts: 1 Cut Endpts: 1 Slack: -0.922 TNS: -0.922 Rel: 1.000
clkB	■	Timed Endpts: 1 Slack: 0.394 TNS: 0.000 Rel: 2.000	■
genClkD	Cut Endpts: 1	■	Timed Endpts: 2 Slack: 0.194 TNS: 0.000 Rel: 1.000

Legend

- ■ No transfers
- 🟦 Clock group
- 🟧 Inactive clocks
- 🟥 Fails timing
- 🟩 Passes timing
- ⬜ Cut transfer

Toggle Data

- ☑ Number of timed endpoints
- ☑ Number of cut endpoints
- ☑ Worst-case slack
- ☑ Endpoint TNS
- ☑ Tightest relationship

☑ Show Filters ☑ Show Legend

Report Panels

The reports consists of:

- Filter boxes for filtering **From Clock:** and **To Clock:** values. Clicking on the **From Clock:** or **To Clock:** opens the **Name Finder** dialog box.
- A color coded grid which displays the clock transfer status. Status colors are defined in the **Legend**.
  - — The clock headers list each clock with transfers in the design. If the name of the clock is too long, the display is truncated, but the full name can be seen in a tool tip or by resizing the clock header cell.
  - — Generated clocks are represented as children of the clock they are derived from. A '+' icon next to a clock name indicates there are generated clocks associated. Clicking on that clock header displays the generated clocks associated with that clock.
- A **Legend** and **Toggle Data** section for controlling the grid display output.
- You can use the **Show Filters** and **Show Legend** controls to turn Filters and Legend on or off.

Transfer Cell Content

Each block in the grid is referred to as a transfer cell. Each transfer cell uses color and text to display important details of the paths involved in a transfer. The color coding represents the following states:

- Black—No transfers. There are no paths crossing between the source and destination clock of this cell.
- Green—Passes timing. All timing paths in this transfer, that have not been cut, meet their timing requirements.
- Red—Fails timing. One or more of the timing paths in the transfer do not meet their timing requirements. If the transfer is between unrelated clocks, the paths likely need to be synchronized by a synchronizer chain.
- Blue—Clock groups. The source and destination clocks of these transfers have been cut by means of asynchronous clock groups.
- Gray—Cut transfer. All paths in this transfer have been cut by false paths. The result of this is that these paths are not considered during timing analysis.
- Orange—Inactive clocks. One of the clocks involved in the transfer has been marked as an inactive clock (with the `set_active_clocks` command). Such transfers are ignored by the Timing Analyzer.

The text in each transfer cell contains data specific to each transfer. Types of data on display can be turned on or off with the **Toggle Data** boxes but you can mouse over any cell to see the full text. These data types are:

- **Number of timed endpoints** between clocks— The number of timed, endpoint-unique paths in the transfer. A path being "timed" means that it was analyzed during timing analysis. Only paths with unique endpoints count towards this total.
- **Number of cut endpoints** between clocks— The number of cut endpoint-unique paths, instead of timed ones. These paths have been cut by either a false path or clock group assignment. Such paths are skipped during timing analysis.
- **Worst-case slack** between clocks— The worst-case slack among all endpoint-unique paths in the transfer.
- **Total negative slack** between clocks— The sum of all negative slacks among all endpoint-unique paths in this transfer.
- **Tightest relationship** between clocks— The lowest-valued setup / hold / recovery / removal relationship between the two clocks in this transfer, depending on the analysis mode of the report

Transfer Cell Operations

Right-click menus allow you to perform operations on transfer cells and clock headers. When the operation is a Timing Analyzer report or SDC command, a dialog box opens prepopulated with the contents of the transfer cell.

Transfer cell operations include:

- **Copy**—Copies the contents of the transfer cell to the clipboard.
- **Copy (include children)**—Copies the name of the chose clock header, and the names of each of its derived clocks. This option only appears for clock headers with generated clocks.
- **Report Timing**—Not available for transfer cells with no valid paths (gray or black cells).
- **Report Endpoints**—Not available for transfer cells with no cut paths (gray or black cells).
- **Report False Path**—Not available for transfer cells with no valid paths (black cells).
- **Report Exceptions**
- **Report Exceptions (with clock groups)**—Only available for clock group transfers (blue cells)
- **Set False Path**
- **Set Multicycle Path**
- **Set Min Delay**
- **Set Max Delay**
- **Set Clock Uncertainty**

Clock header operations include:

- **Copy**—Copies the contents of the clock header to the clipboard.
- **Expand/Collapse All Rows/Columns**—Shows or hides all derived clocks in the grid.
- **Create Slack Histogram**—Generates a slack histogram report for the selected clock.
- **Report Timing From/To Clock**—Generates a timing report for the selected clock. If the clock has not been expanded to display its derived clocks, all clocks derived from the selected clock are included in the timing report as well. To prevent this, expand the clock before right-clicking it.
- **Remove Clock(s)**—Removes the selected clock from the design. If the clock has not been expanded, all clocks derived from the selected clock are also removed.

As with other Timing Analyzer reports, you may view CDC Viewer output in four formats:

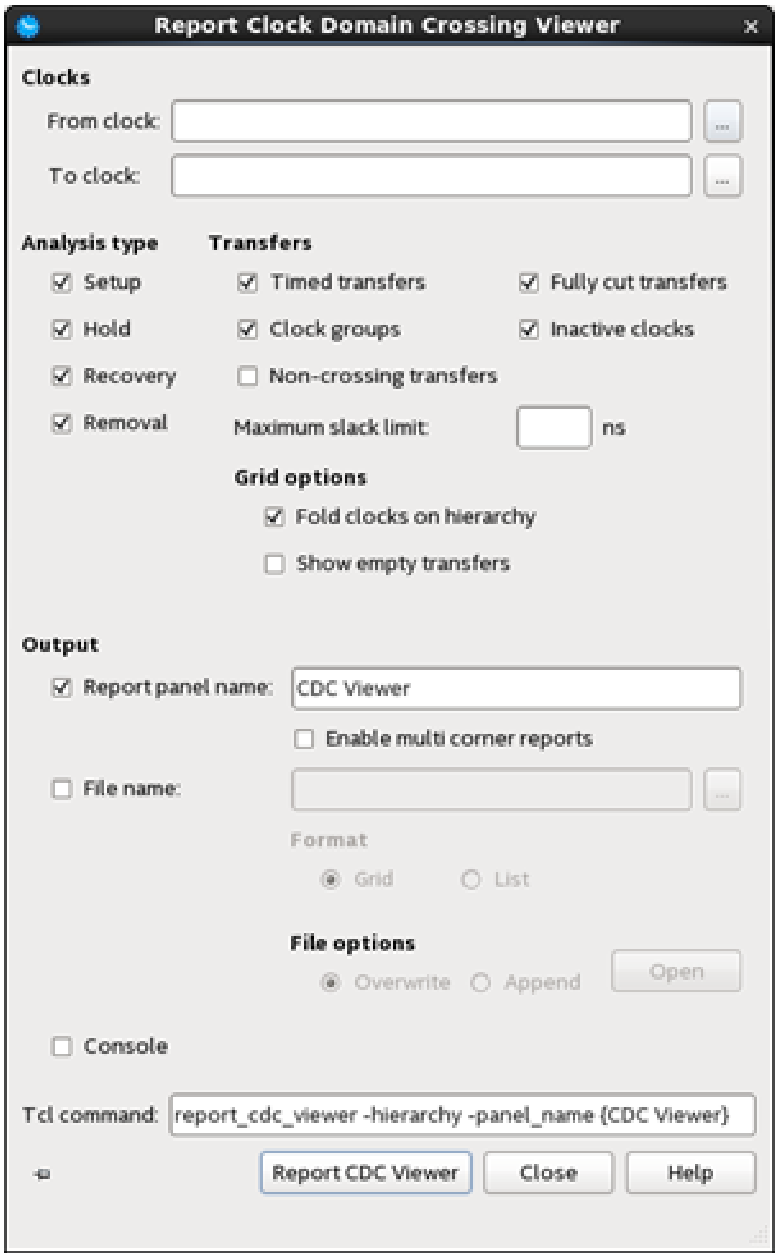
- A report panel in the Timing Analyzer
- Output in the Timing Analyzer Tcl console
- A plain-text file
- An HTML file that can be viewed in a web browser.

The Timing Analyzer report panel is the recommended format.

8.5.4.1 Report Custom CDC Viewer Command

Allows you to configure and display a customized report that creates a customized clock domain crossing report to either a file, the Tcl console, or a report panel. This report displays the results of setup, hold, recovery, and removal checks on clock domain crossing transfers. You open this report in the Timing Analyzer by clicking **Reports ➤ Custom Reports ➤ Report Custom CDC Viewer**.

Note: You can click the **Pushpin** button to keep the **Report False Path**, **Report Timing**, and **Report Endpoints** dialog boxes open after you generate a report. You can use this feature to fine tune your report settings or quickly create additional reports. You can click **Close** to close the dialog box at any time.



Analysis type: [✓ Setup] [✓ Hold] [✓ Recovery] [✓ Removal]
Transfers: [✓ Timed transfers] [✓ Fully cut transfers] [✓ Clock groups] [✓ Inactive clocks] [ ] Non-crossing transfers
Maximum slack limit: ns
Grid options: [✓ Fold clocks on hierarchy] [ ] Show empty transfers
[✓ Report panel name:] CDC Viewer
[ ] Enable multi corner reports
[ ] File name:
Format: [✓ Grid] List
File options: [✓ Overwrite] Append
[ ] Console
Tcl command: report_cdc_viewer -hierarchy -panel_name {CDC Viewer}

Filter boxes for filtering **From Clock:** and **To Clock:** values. Clicking on the buttons next to each box opens the **Name Finder** dialog box.

Analysis Type

The CDC Viewer can analyze any combination of **Setup**, **Hold**, **Recovery**, or **Removal**.

Scripting Information
Keyword: `report_cdc_viewer` **Settings:** `-setup|-hold|-recovery|-removal`

Transfer Filtering

By default, all transfer types are included in CDC Viewer reports:

- **Timed transfers**—Passing or failing
- **Fully cut transfers**— Transfers where all paths are false paths.
- **Clock groups**
- **Inactive clocks**

You can use these options to narrow down which kinds of transfers to show, or by adding the desired transfer types as options: –timed, –fully_cut, –clock_groups, and –inactive. If none are specified, all transfer types are shown.

Scripting Information
Keyword: `report_cdc_viewer` **Settings:** `-timed|-fully_cut|-clock_groups|-inactive`

By default, only clocks are shown are clocks that launch or latch paths that are launched or latched by clocks other than themselves. Turning on **Non-crossing transfers** shows clocks with transfers to or from themselves.

Scripting Information
Keyword: `report_cdc_viewer` **Settings:** `-show_non_crossing`

Note: In grid-formatted reports, clocks with non-crossing transfers are always shown as long as they have transfers between other clocks too.

If you specify a value in the **Maximum slack limit** box, only paths with slack less than the value are displayed. If this option is not included, paths of any slack value are included in the report.

Scripting Information
Keyword: `report_cdc_viewer` **Settings:** `-less_than_slack`

Grid Options

In grid-formatted reports, the grid can be configured to display clocks as either a flat list or in a hierarchy where generated clocks are displayed as children of the clock they are derived from. Turn on **Fold clocks on hierarchy** to enable this nested display.

Scripting Information
Keyword: `report_cdc_viewer` **Settings:** `-hierarchy`

By default, clocks that launch or latch to no paths are not shown in grid-based reports. You can show these clocks by turning on the **Show empty transfers** option.

Scripting Information
Keyword: `report_cdc_viewer` **Settings:** `-show_empty`

Output

Allows you to specify where you want to save the report and how much detail you want in the report. You can select one or more of the following settings:

- **Report panel name**— Directs the Timing Analyzer to generate a report panel with the specified name. The default report name is Report Timing.

Scripting Information
Keyword: `report_cdc_viewer` **Settings:** `-panel_name<reportname>`

- **Enable multi-corner reports**— Allows you to enable or disable multi-corner timing analysis. This option is on by default.

Scripting Information
Keyword: `report_cdc_viewer` **Settings:** `-multi_corner`

- **File name**— Directs the Timing Analyzer to save the report to your local disk as a text file with the specified file name. To save a report in HTML, end the filename with ".html".

Scripting Information
Keyword: `report_cdc_viewer` **Settings:** `-file<filename>`

- **Format**— Specifies that the generated report file is formatted as a list of clock transfers rather than the default grid panel.

Scripting Information
Keyword: `report_cdc_viewer` **Settings:** `-list`

- Under **File options** you can specify whether the Timing Analyzer overwrites an existing file (the default setting) or appends the content to an existing file.

Scripting Information
Keyword: `report_cdc_viewer` **Settings:** `-append|-overwrite`

- **Console**— Specifies whether the report appears as information messages in the **Console**.

Scripting Information
Keyword: `report_cdc_viewer` **Settings:** `-stdout`

8.5.5 Analyzing Results with Report Timing

Report Timing is one of the most useful analysis tools in Timing Analyzer. Many designs require nothing but this command. In the Timing Analyzer, this command can be accessed from the **Tasks** menu , from the **Reports ➤ Custom Reports** menu, or by right-clicking on nodes or assignments in Timing Analyzer.

You can review all of the options for **Report Timing** by typing `report_timing -long_help` in the Timing Analyzer console.

Clocks

The **From Clock** and **To Clock** in the **Clocks** box are used to filter paths where the selected clock is used as the launch or latch. The pull-down menu allows you to choose from existing clocks (although admittedly has a "limited view" for long clock names).

Targets

The boxes in the **Targets** box are targets for the **From Clock** and **To Clock**settings, and allow you to report paths with only particular endpoints. These are usually filled with register names or I/O ports, and can be wildcarded. For example, you might use the following to only report paths within a hierarchy of interest:

```
report_timing -from *|egress:egress_inst|* -to *|
egress:egress_inst|* -(other options)
```

If the **From**, **To**, or **Through** boxes are empty, then the Timing Analyzer assumes you are refering to all possible targets in the device, which can also be represented with a wildcard (*). The **From** and **To** options cover the majority of situations. The**Through** option is used to limit the report for paths that pass through combinatorial logic, or a particular pin on a cell. This is seldom used, and may not be very reliable due to combinatorial node name changes during synthesis. Clicking the browse **Browse** box after each target opens the **Name Finder** dialog box to search for specific names. This is especially useful to make sure the name being entered matches nodes in the design, since the **Name Finder** can immediately show what matches a user's wildcard.

Analysis type

The **Analysis type** options are **Setup**, **Hold**, **Recovery**, or **Removal**.

Output

The **Detail** level, has the following options:

The first level is called **Summary**, and produces a report which only displays summary information such as:

- **Slack**
- **From Node**
- **To Node**
- **Launch Clock**
- **Latch Clock**
- **Relationship**
- **Clock Skew**
- **Data Delay**

The **Summary** report is always reported with more detailed reports, so the user would choose this if they want less info. A good use for summary detail is when writing the report to a text file, where **Summary** can be quite brief.

The next level is **Path only**. This report displays all the detailed information, except the **Data Path** tab displays the clock tree as one line item. This is useful when you know the clock tree is correct, details are not relevant. This is common for most paths within the FPGA. A useful data point is to look at the **Clock Skew** column in the **Summary** report, and if it's a small number, say less than +/-150ps, then the clock tree is well balanced between source and destination.

If there is clock skew, you should select the **Full path** option.. This breaks the clock tree out into explicit detail, showing every cell it goes through, including such things as the input buffer, PLL, global buffer (called CLKCTRL_), and any logic. If there is clock skew, this is where you can determine what is causing the clock skew in your design. The **Full path** option is also recommended for I/O analysis, since only the source clock or destination clock is inside the FPGA, and therefore its delay plays a critical role in meeting timing.

The Data Path tab of a detailed report gives the delay break-downs, but there is also useful information in the **Path Summary** and **Statistics** tabs, while the **Waveform** tab is useful to help visualize the **Data Path** analysis. I would suggest taking a few minutes to look at these in the user's design. The whole analysis takes some time to get comfortable with, but hopefully is clear in what it's doing.

Enable multi corner reports allows you to enable or disable multi-corner timing analysis. This option is on by default.

Report Timing also has the **Report panel name**, which displays the name used in Timing Analyzer's **Report** section. There is also an optional **File name** switch, which allows you to write the information to a file. If you append .htm as a suffix, the Timing Analyzer produces the report as HTML. The **File options** radio buttons allow you to choose between **Overwrite** and **Append** when saving the file.

Paths

The default value for **Report number of paths** is 10. Two endpoints may have a lot of combinatorial logic between them and might have many different paths. Likewise, a single destination may have hundreds of paths leading to it. Because of this, you might list hundreds of paths, many of which have the same destination and might have the same source. By turning on **Pairs only** you can list only one path for each pair of source and destination. An even more powerful way to filter the report is limit the Maximum number of paths per endpoints. You can also filter paths by entering a value in the **Maximum slack limit** field.

Tcl command

Finally, at the bottom is the **Tcl command**field, which displays the Tcl syntax of what is run in Timing Analyzer. You can edit this directly before running the **Report Timing** command.

Note:

A useful addition is to addis the `-false_path` option to the command line string. With this option, only false paths are listed. A false path is any path where the launch and latch clock have been defined, but the path was cut with either a *`set_false_path`* assignment or *`set_clock_groups_assignment`*. Paths where the launch or latch clock was never constrained are not considered false paths. This option is useful to see if a false path assignment worked and what paths it covers, or to look for paths between clock domains that should not exist. The **Task** window's **Report False Path** custom report is nothing more than **Report Timing** with the `-false_path` flag enabled.

8.5.6 Correlating Constraints to the Timing Report

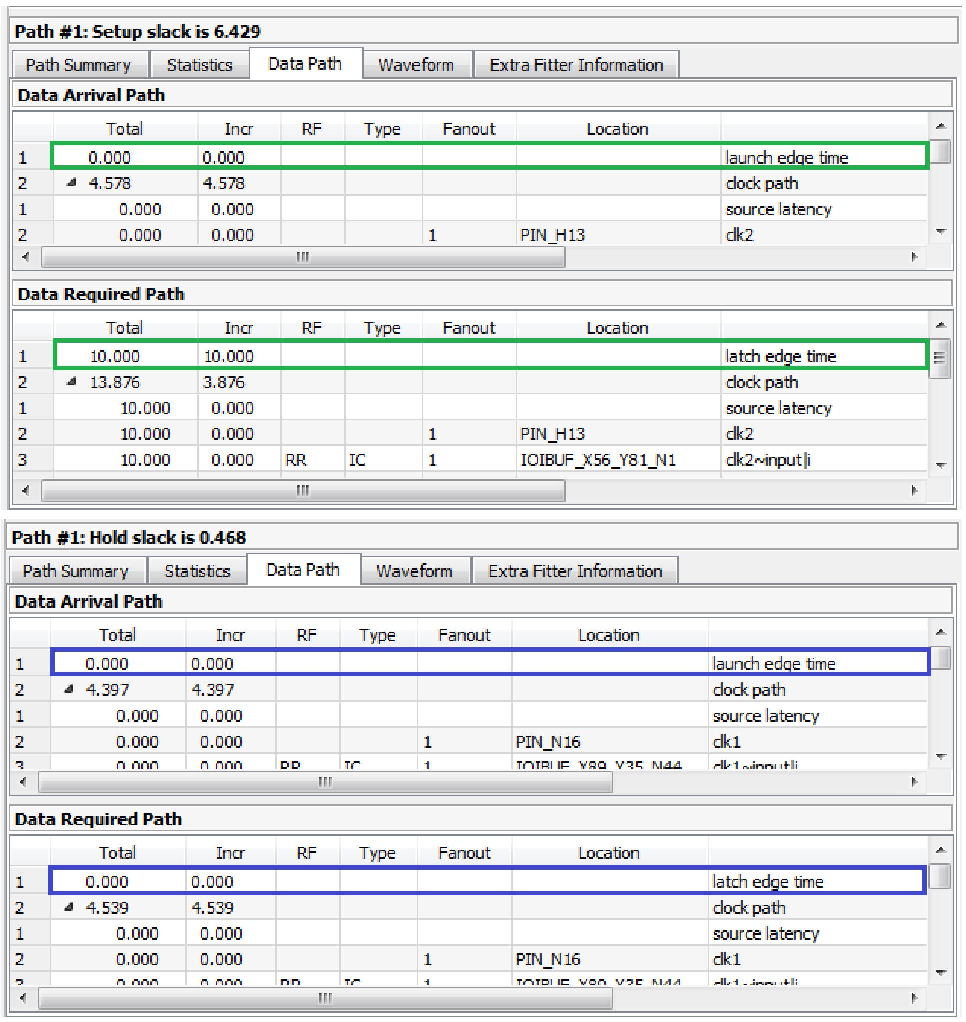
A critical part of timing analysis is how timing constraints appear in the **Report Timing** analysis. Most constraints only affect the launch and latch edges. Specifically, `create_clock` and `create_generated_clock` create clocks with default relationships. The command `set_multicycle_path` modifies those default relationships, while `set_max_delay` and `set_min_delay` are low-level overrides that explicitly tell Timing Analyzer what the launch and latch edges should be.

The following figures are from an example of the output of **Report Timing** on a particular path.

Initially, the design features a clock driving the source and destination registers with a period of 10ns. This results in a setup relationship of 10ns (launch edge = 0ns, latch edge = 10ns) and hold relationship of 0ns (launch edge = 0ns, latch edge = 0ns) from the command:

```
create_clock -name clocktwo -period 10.000 [get_ports {clk2}]
```

Figure 109. Setup Relationship 10ns, Hold Relationship 0ns



Path #1: Setup slack is 6.429

Path Summary | Statistics | Data Path | Waveform | Extra Fitter Information

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	
1	0.000	0.000					launch edge time
2	◢ 4.578	4.578					clock path
1	0.000	0.000					source latency
2	0.000	0.000			1	PIN_H13	clk2

Data Required Path

	Total	Incr	RF	Type	Fanout	Location	
1	10.000	10.000					latch edge time
2	◢ 13.876	3.876					clock path
1	10.000	0.000					source latency
2	10.000	0.000			1	PIN_H13	clk2
3	10.000	0.000	RR	IC	1	IOIBUF_X56_Y81_N1	clk2~input|i

Path #1: Hold slack is 0.468

Path Summary | Statistics | Data Path | Waveform | Extra Fitter Information

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	
1	0.000	0.000					launch edge time
2	◢ 4.397	4.397					clock path
1	0.000	0.000					source latency
2	0.000	0.000			1	PIN_N16	clk1
3	0.000	0.000	RR	IC	1	IOIBUF_X89_Y35_N44	clk1~input|i

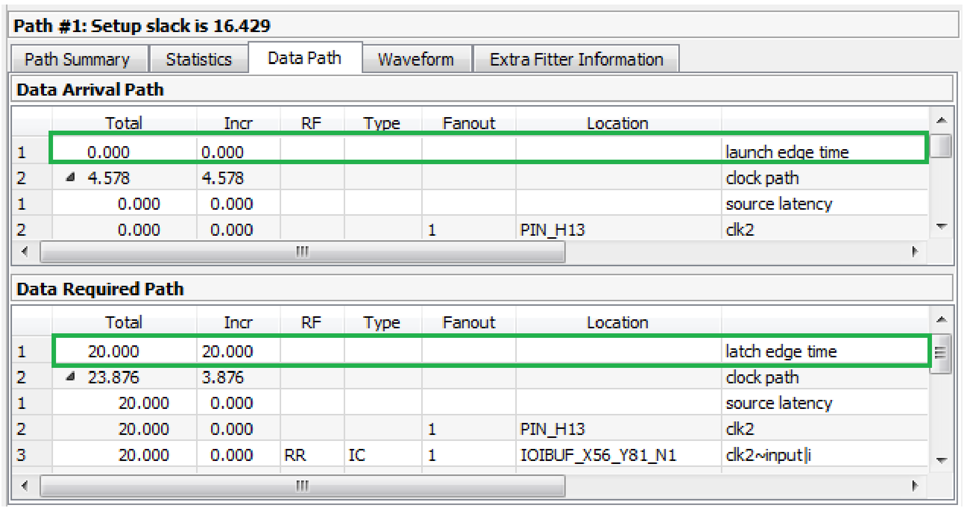
Data Required Path

	Total	Incr	RF	Type	Fanout	Location	
1	0.000	0.000					latch edge time
2	◢ 4.539	4.539					clock path
1	0.000	0.000					source latency
2	0.000	0.000			1	PIN_N16	clk1
3	0.000	0.000	RR	IC	1	IOIBUF_X89_Y35_N44	clk1~input|i

In the next figure, using `set_multicycle_path` adds multicycles to relax the setup relationship, or open the window, making the setup relationship 20ns while the hold relationship is still 0ns:

```
set_multicycle_path -from clocktwo -to clocktwo -setup -end 2
set_multicycle_path -from clocktwo -to clocktwo -hold -end 1
```

Figure 110. Setup Relationship 20ns



Path #1: Setup slack is 16.429

Path Summary | Statistics | Data Path | Waveform | Extra Fitter Information

Data Arrival Path

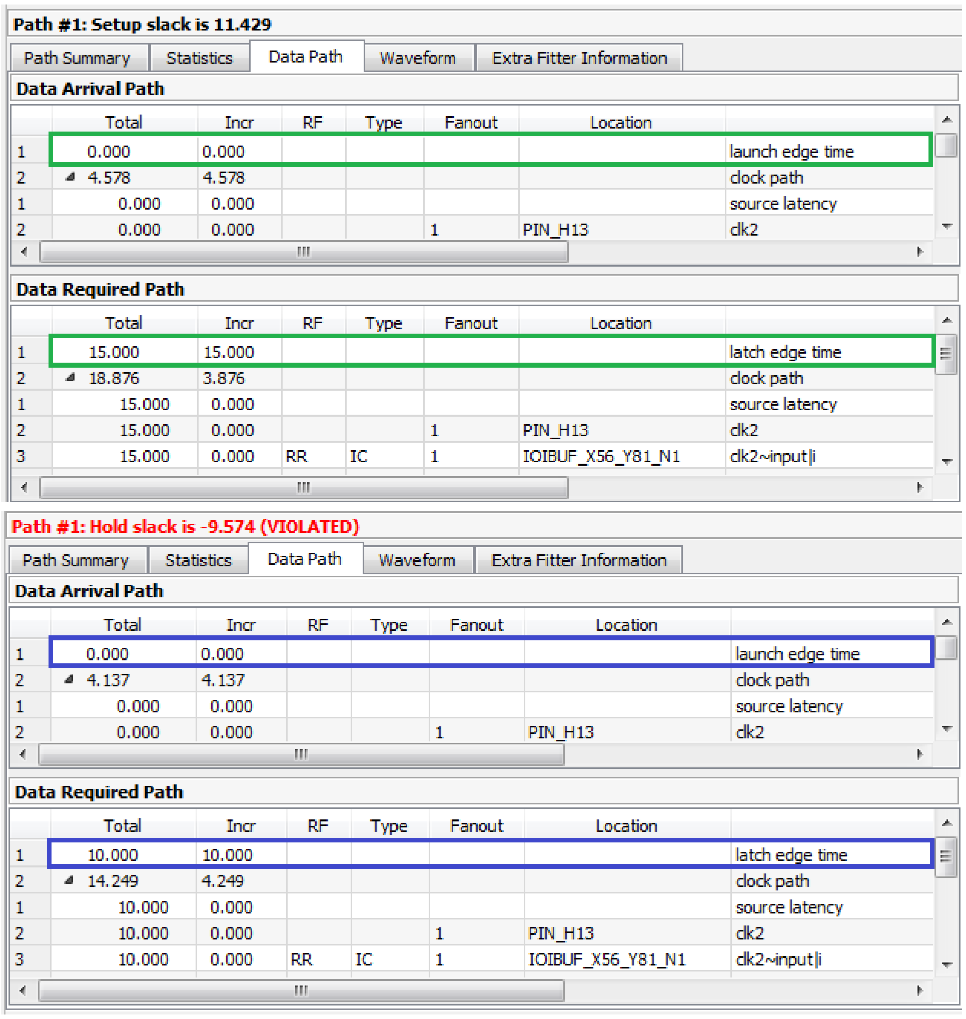
	Total	Incr	RF	Type	Fanout	Location	
1	0.000	0.000					launch edge time
2	◢ 4.578	4.578					clock path
1	0.000	0.000					source latency
2	0.000	0.000			1	PIN_H13	clk2

Data Required Path

	Total	Incr	RF	Type	Fanout	Location	
1	20.000	20.000					latch edge time
2	◢ 23.876	3.876					clock path
1	20.000	0.000					source latency
2	20.000	0.000			1	PIN_H13	clk2
3	20.000	0.000	RR	IC	1	IOIBUF_X56_Y81_N1	clk2~input|i

In the last figure, using the `set_max_delay` and `set_min_delay` constraints lets you explicitly override the relationships. Note that the only thing changing for these different constraints is the Launch Edge Time and Latch Edge Times for setup and hold analysis. Every other line item comes from delays inside the FPGA and are static for a given fit. Whenever analyzing how your constraints affect the timing requirements, this is the place to look.

Figure 111. Using set_max_delay and set_min_delay



Path #1: Setup slack is 11.429

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	
1	0.000	0.000					launch edge time
2	4.578	4.578					clock path
1	0.000	0.000					source latency
2	0.000	0.000			1	PIN_H13	clk2

Data Required Path

	Total	Incr	RF	Type	Fanout	Location	
1	15.000	15.000					latch edge time
2	18.876	3.876					clock path
1	15.000	0.000					source latency
2	15.000	0.000			1	PIN_H13	clk2
3	15.000	0.000	RR	IC	1	IOIBUF_X56_Y81_N1	clk2~input|i

Path #1: Hold slack is -9.574 (VIOLATED)

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	
1	0.000	0.000					launch edge time
2	4.137	4.137					clock path
1	0.000	0.000					source latency
2	0.000	0.000			1	PIN_H13	clk2

Data Required Path

	Total	Incr	RF	Type	Fanout	Location	
1	10.000	10.000					latch edge time
2	14.249	4.249					clock path
1	10.000	0.000					source latency
2	10.000	0.000			1	PIN_H13	clk2
3	10.000	0.000	RR	IC	1	IOIBUF_X56_Y81_N1	clk2~input|i

For I/O, this all holds true except we must add in the `-max` and `-min` values. They are displayed as **iExt** or **oExt** in the **Type** column. An example would be an output port with a `set_output_delay -max 1.0` and `set_output_delay -min -0.5`:

Once again, the launch and latch edge times are determined by the clock relationships, multicycles and possibly `set_max_delay` or `set_min_delay` constraints. The value of `set_output_delay` is also added in as an **oExt** value. For outputs this value is part of the **Data Required Path**, since this is the external part of the analysis. The setup report on the left subtracts the `-max` value, making the setup relationship harder to meet, since we want the **Data Arrival Path** to be shorter than the **Data Required Path**. The `-min` value is also subtracted, which is why a negative number makes hold timing more restrictive, since we want the **Data Arrival Path** to be longer than the **Data Required Path**.

Related Links

Relaxing Setup with set_multicyle_path on page 111

8.6 Constraining and Analyzing with Tcl Commands

You can use Tcl commands from the Intel Quartus Prime software Tcl Application Programming Interface (API) to constrain, analyze, and collect information for your design. This section focuses on executing timing analysis tasks with Tcl commands; however, you can perform many of the same functions in the Timing Analyzer GUI. SDC commands are Tcl commands for constraining a design. SDC extension commands provide additional constraint methods and are specific to the Timing Analyzer. Additional Timing Analyzer commands are available for controlling timing analysis and reporting. These commands are contained in the following Tcl packages available in the Intel Quartus Prime software:

- `::quartus::sta`
- `::quartus::sdc`
- `::quartus::sdc_ext`

Related Links

- ::quartus::sta
  For more information about Timing Analyzer Tcl commands and a complete list of commands, refer to Intel Quartus Prime Help.
- ::quartus::sdc
  For more information about standard SDC commands and a complete list of commands, refer to Intel Quartus Prime Help.
- ::quartus::sdc_ext
  For more information about Intel FPGA extensions of SDC commands and a complete list of commands, refer to Intel Quartus Prime Help.

8.6.1 Collection Commands

The Timing Analyzer Tcl commands often return data in an object called a collection. In your Tcl scripts you can iterate over the values in collections to access data contained in them. The software returns collections instead of Tcl lists because collections are more efficient than lists for large sets of data.

The Timing Analyzer supports collection commands that provide easy access to ports, pins, cells, or nodes in the design. Use collection commands with any constraints or Tcl commands specified in the Timing Analyzer.

Table 33. SDC Collection Commands

Command	Description of the collection returned
`all_clocks`	All clocks in the design.
`all_inputs`	All input ports in the design.
`all_outputs`	All output ports in the design.
`all_registers`	All registers in the design.
`get_cells`	Cells in the design. All cell names in the collection match the specified pattern. Wildcards can be used to select multiple cells at the same time.
`get_clocks`	Lists clocks in the design. When used as an argument to another command, such as the `-from` or `-to` of `set_multicycle_path`, each node in the clock represents all nodes clocked by the clocks in the collection. The default uses the specific node (even if it is a clock) as the target of a command.

continued...

Command	Description of the collection returned
`get_nets`	Nets in the design. All net names in the collection match the specified pattern. You can use wildcards to select multiple nets at the same time.
`get_pins`	Pins in the design. All pin names in the collection match the specified pattern. You can use wildcards to select multiple pins at the same time.
`get_ports`	Ports (design inputs and outputs) in the design.

You can also examine collections and experiment with collections using wildcards in the Timing Analyzer by clicking **Name Finder** from the **View** menu.

8.6.1.1 Wildcard Characters

To apply constraints to many nodes in a design, use the "`*`" and "?" wildcard characters. The "`*`" wildcard character matches any string; the "?" wildcard character matches any single character.

If you make an assignment to node `reg*`, the Timing Analyzer searches for and applies the assignment to all design nodes that match the prefix `reg` with any number of following characters, such as `reg`, `reg1`, `reg[2]`, `regbank`, and `reg12bank`.

If you make an assignment to a node specified as `reg?`, the Timing Analyzer searches and applies the assignment to all design nodes that match the prefix `reg` and any single character following; for example, `reg1`, `rega`, and `reg4`.

8.6.1.2 Adding and Removing Collection Items

Wildcards used with collection commands define collection items identified by the command. For example, if a design contains registers named `src0`, `src1`, `src2`, and `dst0`, the collection command `[get_registers src*]` identifies registers `src0`, `src1`, and `src2`, but not register `dst0`. To identify register `dst0`, you must use an additional command, `[get_registers dst*]`. To include `dst0`, you could also specify a collection command `[get_registers {src* dst*}]`.

To modify collections, use the `add_to_collection` and `remove_from_collection` commands. The `add_to_collection` command allows you to add additional items to an existing collection.

add_to_collection Command

add_to_collection <*first collection*> <*second collection*>

Note: The `add_to_collection` command creates a new collection that is the union of the two specified collections.

The `remove_from_collection` command allows you to remove items from an existing collection.

remove_from_collection Command

remove_from_collection <*first collection*> <*second collection*>

You can use the following code as an example for using `add_to_collection` for adding items to a collection.

Adding Items to a Collection

```
#Setting up initial collection of registers
set regs1 [get_registers a*]
#Setting up initial collection of keepers
set kprs1 [get_keepers b*]
#Creating a new set of registers of $regs1 and $kprs1
set regs_union [add_to_collection $kprs1 $regs1]
#OR
#Creating a new set of registers of $regs1 and b*
#Note that the new collection appends only registers with name b*
# not all keepers
set regs_union [add_to_collection $regs1 b*]
```

In the Intel Quartus Prime software, keepers are I/O ports or registers. A SDC file that includes `get_keepers` can only be processed as part of the Timing Analyzer flow and is not compatible with third-party timing analysis flows.

Related Links

- add_to_collection
- remove_from_collection
  For more information about the`add_to_collection` and `remove_from_collection` commands, refer to Intel Quartus Prime Help.

8.6.1.3 Getting Other Information about Collections

You can display the contents of a collection with the `query_collection` command. Use the `-report_format` option to return the contents in a format of one element per line. The `-list_format` option returns the contents in a Tcl list.

```
query_collection -report_format -all $regs_union
```

Use the `get_collection_size` command to return the size of a collection; the number of items it contains. If your collection is in a variable named `col`, it is more efficient to use `set num_items [get_collection_size $col]` than `set num_items [llength [query_collection -list_format $col]]`

8.6.1.4 Using the get_pins Command

The `get_pins` command supports options that control the matching behavior of the wildcard character (*). Depending on the combination of options you use, you can make the wildcard character (*) respect or ignore individual levels of hierarchy, which are indicated by the pipe character (|). By default, the wildcard character (*) matches only a single level of hierarchy.

These examples filter the following node and pin names to illustrate function:

- foo (a hierarchy level named foo)
- foo|dataa (an input pin in the instance foo)
- foo|datab (an input pin in the instance foo)
- foo|bar (a combinational node named bar in the foo instance)
- foo|bar|datac (an input pin to the combinational node named bar)
- foo|bar|datad (an input pin to the combinational node bar)

Table 34. Sample Search Strings and Search Results

Search String	Search Result
`get_pins *|dataa`	`foo|dataa`
`get_pins *|datac`	*<empty>*(3)
`get_pins *|*|datac`	`foo|bar|datac`
`get_pins foo*|*`	`foo|dataa, foo|datab`
`get_pins -hierarchical *|*|datac`	*<empty>*(3)
`get_pins -hierarchical foo|*`	`foo|dataa, foo|datab`
`get_pins -hierarchical *|datac`	`foo|bar|datac`
`get_pins -hierarchical foo|*|datac`	*<empty>*(3)
`get_pins -compatibility_mode *|datac`	`foo|bar|datac` (4)
`get_pins -compatibility_mode *|*|datac`	`foo|bar|datac`

The default method separates hierarchy levels of instances from nodes and pins with the pipe character (|). A match occurs when the levels of hierarchy match, and the string values including wildcards match the instance and/or pin names. For example, the command `get_pins <`*`instance_name`*`>|*|datac` returns all the `datac` pins for registers in a given instance. However, the command `get_pins *|datac` returns and empty collection because the levels of hierarchy do not match.

Use the `-hierarchical` matching scheme to return a collection of cells or pins in all hierarchies of your design.

For example, the command `get_pins -hierarchical *|datac` returns all the `datac` pins for all registers in your design. However, the command `get_pins -hierarchical *|*|datac` returns an empty collection because more than one pipe character (|) is not supported.

The `-compatibility_mode` option returns collections matching wildcard strings through any number of hierarchy levels. For example, an asterisk can match a pipe character when using `-compatibility_mode`.

8.6.2 Identifying the Intel Quartus Prime Software Executable from the SDC File

To identify which Intel Quartus Prime software executable is currently running you can use the `$::TimeQuestInfo(nameofexecutable)` variable from within an SDC file. This technique is most commonly used when you want to use an overconstraint to cause the Fitter to work harder on a particular path or set of paths in the design.

(3) The search result is *<empty>* because the wildcard character (*) does not match more than one hierarchy level, indicated by a pipe character (|), by default. This command would match any pin named `datac` in instances at the top level of the design.

(4) When you use `-compatibility_mode`, pipe characters (|) are not treated as special characters when used with wildcards.

Identifying the Intel Quartus Prime Executable

```
#Identify which executable is running:
set current_exe $::TimeQuestInfo(nameofexecutable)
if { [string equal $current_exe "quartus_fit"] } {
    #Apply .sdc assignments for Fitter executable here
} else {
    #Apply .sdc assignments for non-Fitter executables here
}
if { ! [string equal "quartus_sta" $::TimeQuestInfo(nameofexecutable)] } {
    #Apply .sdc assignments for non-TimeQuest executables here
} else {
    #Apply .sdc assignments for TimeQuest executable here
}
```

Examples of different executable names are `quartus_map` for Analysis & Synthesis, `quartus_fit` for Fitter, and `quartus_sta` for the Timing Analyzer.

8.6.3 Locating Timing Paths in Other Tools

You can locate paths and elements from the Timing Analyzer to other tools in the Intel Quartus Prime software.

Use the **Locate** or `Locate Path` command in the Timing Analyzer GUI or the `locate` command in the Tcl console in the Timing Analyzer GUI. Right-click most paths or node names in the Timing Analyzer GUI to access the **Locate** or **Locate Path** options.

The following commands are examples of how to locate the ten paths with the worst timing slack from Timing Analyzer to the **Technology Map Veiwer** and locate all ports matching data* in the **Chip Planner**.

Example 13. Locating from the Timing Analyzer

```
# Locate in the Technology Map Viewer the ten paths with the worst slack
locate [get_timing_paths -npaths 10] -tmv
# locate all ports that begin with data in the Chip Planner
locate [get_ports data*] -chip
```

Related Links

locate
    For more information on this command, refer to Intel Quartus Prime Help.

8.7 Generating Timing Reports

The Timing Analyzer provides real-time static timing analysis result reports. The Timing Analyzer does not automatically generate most reports; you must create each report individually in the Timing Analyzer GUI or with command-line commands. You can customize in which report to display specific timing information, excluding fields that are not required.

Some of the different command-line commands you can use to generate reports in the Timing Analyzer and the equivalent reports shown in the Timing Analyzer GUI.

Table 35. Timing Analyzer Reports

Command-Line Command	Report
`report_timing`	Timing report
`report_exceptions`	Exceptions report
`report_clock_transfers`	Clock Transfers report
`report_min_pulse_width`	Minimum Pulse Width report
`report_ucp`	Unconstrained Paths report

During compilation, the Intel Quartus Prime software generates timing reports on different timing areas in the design. You can configure various options for the Timing Analyzer reports generated during compilation.

You can also use the `TIMEQUEST_REPORT_WORST_CASE_TIMING_PATHS` assignment to generate a report of the worst-case timing paths for each clock domain. This report contains worst-case timing data for setup, hold, recovery, removal, and minimum pulse width checks.

Use the `TIMEQUEST_REPORT_NUM_WORST_CASE_TIMING_PATHS` assignment to specify the number of paths to report for each clock domain.

An example of how to use the `TIMEQUEST_REPORT_WORST_CASE_TIMING_PATHS` and `TIMEQUEST_REPORT_NUM_WORST_CASE_TIMING_PATHS` assignments in the `.qsf` to generate reports.

For more information about the options you can set to customize Timing Analyzer reports, refer to the Timing Analyzer page in Intel Quartus Prime Help.

For more information about timing closure recommendations, refer to the *Timing Closure and Optimization* page of the *Intel Quartus Prime Handbook, Volume 2*.

Generating Worst-Case Timing Reports

```
# Enable Worst-Case Timing Report
set_global_assignment -name TIMEQUEST_REPORT_WORST_CASE_TIMING_PATHS ON
# Report 10 paths per clock domain
set_global_assignment -name TIMEQUEST_REPORT_NUM_WORST_CASE_TIMING_PATHS 10
```

Fmax Summary Report panel

The Fmax Summary Report panel lists the maximum frequency of each clock in your design. In some designs you may see a note indicating "Limit due to hold check. Typically, Fmax is not limited by hold checks, because they are often same-edge relationships, and therefore independent of clock frequency, for example, launch = 0, latch = 0. However, if you have an inverted clock transfer, or a multicycle transfer such as setup=2, hold=0, then the hold relationship is no longer a same-edge transfer and changes as the clock frequency changes. The value in the **Restricted Fmax** column incorporates limits due to hold time checks in the situations described previously, as well as minimum period and pulse width checks. If hold checks limit the Fmax more than setup checks, that is indicated in the **Note:** column as "Limit due to hold check".

Related Links

- ::quartus::sta
  In Intel Quartus Prime Help
- Timing Analyzer Page
  For more information about the options you can set to customize Timing Analyzer reports.
- Timing Closure and Optimization
  In *Intel Quartus Prime Standard Edition Handbook Volume 2*

8.8 Document Revision History

Table 36. Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	• Changed instances of *Quartus II* to *Quartus Prime*. • Updated information on using Intel Arria 10 devices with enhanced timing algorithms.
2015.05.04	15.0.0	Added and updated contents in support of new timing algorithms for Arria 10: • Enhanced Timing Analysis for Arria 10 • Maximum Skew (`set_max_skew` command) • Net Delay (`set_net_delay` command) • Create Generated Clocks (clock-as-data example)
2014.12.15	14.1.0	Major reorganization. Revised and added content to the following topic areas: • Timing Constraints • Create Clocks and Clock Constraints • Creating Generated Clocks • Creating Clock Groups • Clock Uncertainty • Running the Timing Analyzer • Generating Timing Reports • Understanding Results • Constraining and Analyzing with Tcl Commands
August 2014	14.0a10.0	Added command line compliation requirements for Arria 10 devices.
June 2014	14.0.0	• Minor updates. • Updated format.
November 2013	13.1.0	• Removed HardCopy device information.
June 2012	12.0.0	• Reorganized chapter. • Added “Creating a Constraint File from Intel Quartus Prime Templates with the Intel Quartus Prime Text Editor” section on creating an SDC constraints file with the **Insert Template** dialog box. • Added “Identifying the Intel Quartus Prime Software Executable from the SDC File” section. • Revised multicycle exceptions section.
November 2011	11.1.0	• Consolidated content from the Best Practices for the Intel Quartus Prime Timing Analyzer chapter. • Changed to new document template.
May 2011	11.0.0	• Updated to improve flow. Minor editorial updates.
December 2010	10.1.0	• Changed to new document template. • Revised and reorganized entire chapter. • Linked to Intel Quartus Prime Help.

continued...

Date	Version	Changes
July 2010	10.0.0	Updated to link to content on SDC commands and the Timing Analyzer GUI in Intel Quartus Prime Help.
November 2009	9.1.0	Updated for the Intel Quartus Prime software version 9.1, including: • Added information about commands for adding and removing items from collections • Added information about the set_timing_derate and report_skew commands • Added information about worst-case timing reporting • Minor editorial updates
November 2008	8.1.0	Updated for the Intel Quartus Prime software version 8.1, including: • Added the following sections: "set_net_delay" on page 7–42 "Annotated Delay" on page 7–49 "report_net_delay" on page 7–66 • Updated the descriptions of the `-append` and `-file` <*name*> options in tables throughout the chapter • Updated entire chapter using 8½" × 11" chapter template • Minor editorial updates

Related Links

Documentation Archive

For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

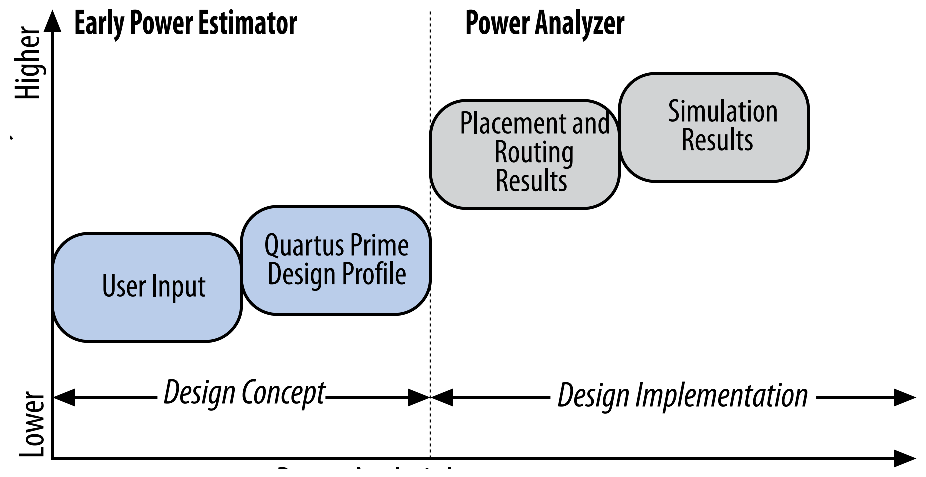
9 Power Analysis

The Intel Quartus Prime Power Analysis tools allow you to estimate device power consumption accurately.

As designs grow larger and process technology continues to shrink, power becomes an increasingly important design consideration. When designing a PCB, you must estimate the power consumption of a device accurately to develop an appropriate power budget, and to design the power supplies, voltage regulators, heat sink, and cooling system.

The following figure shows the Power Analysis tools ability to estimate power consumption from early design concept through design implementation.

Figure 112. Power Analysis From Design Concept Through Design Implementation



For the majority of the designs, the Power Analyzer and the EPE spreadsheet have the following accuracy after the power models are final:

- Power Analyzer—±20% from silicon, assuming that the Power Analyzer uses the Value Change Dump File (`.vcd`) generated toggle rates.
- EPE spreadsheet— ±20% from the Power Analyzer results using `.vcd` generated toggle rates. 90% of EPE designs (using `.vcd` generated toggle rates exported from PPPA) are within ±30% silicon.

The toggle rates are derived using the Power Analyzer with a `.vcd` file generated from a gate level simulation representative of the system operation.

9.1 Types of Power Analyses

Understanding the uses of power analysis and the factors affecting power consumption helps you to use the Power Analyzer effectively. Power analysis meets the following significant planning requirements:

- **Thermal planning**—Thermal power is the power that dissipates as heat from the FPGA. You must use a heatsink or fan to act as a cooling solution for your device. The cooling solution must be sufficient to dissipate the heat that the device generates. The computed junction temperature must fall within normal device specifications.
- **Power supply planning**—Power supply is the power needed to run your device. Power supplies must provide adequate current to support device operation.

  Note: For power supply planning, use the EPE at the early stages of your design cycle. Use the Power Analyzer reports when your design is complete to get an estimate of your design power requirement.

The two types of analyses are closely related because much of the power supplied to the device dissipates as heat from the device; however, in some situations, the two types of analyses are not identical. For example, if you use terminated I/O standards, some of the power drawn from the power supply of the device dissipates in termination resistors rather than in the device.

Power analysis also addresses the activity of your design over time as a factor that impacts the power consumption of the device. The static power (P_{STATIC}) is the thermal power dissipated on chip, independent of user clocks. P_{STATIC} includes the leakage power from all FPGA functional blocks, except for I/O DC bias power and transceiver DC bias power, which are accounted for in the I/O and transceiver sections. Dynamic power is the additional power consumption of the device due to signal activity or toggling.

9.1.1 Differences between the EPE and the Intel Quartus Prime Power Analyzer

The following table lists the differences between the EPE and the Intel Quartus Prime Power Analyzer.

Table 37. Comparison of the EPE and Intel Quartus Prime Power Analyzer

Characteristic	EPE	Intel Quartus Prime Power Analyzer
Phase in the design cycle	Any time, but it is recommended to use Intel Quartus Prime Power Analyzer for post-fit power analysis.	Post-fit
Tool requirements	Spreadsheet program	The Intel Quartus Prime software
Accuracy	Medium	Medium to very high
Data inputs	• Resource usage estimates • Clock requirements • Environmental conditions • Toggle rate	• Post-fit design • Clock requirements • Signal activity defaults • Environmental conditions

continued...

Characteristic	EPE	Intel Quartus Prime Power Analyzer
		• Register transfer level (RTL) simulation results (optional) • Post-fit simulation results (optional) • Signal activities per node or entity (optional)
Data outputs	• Total thermal power dissipation • Thermal static power • Thermal dynamic power • Off-chip power dissipation • Current drawn from voltage supplies	• Total thermal power • Thermal static power • Thermal dynamic power • Thermal I/O power • Thermal power by design hierarchy • Thermal power by block type • Thermal power dissipation by clock domain • Off-chip (non-thermal) power dissipation • Device supply currents

The result of the Power Analyzer is only an estimation of power. Intel FPGA does not recommend using the result as a specification. The purpose of the estimation is to help you establish guidelines for the power budget of your design. It is important that you verify the actual power during device operation as the information is sensitive to the actual device design and the environmental operating conditions.

Note: The Power Analyzer does not include the transceiver power for features that can only be enabled through dynamic reconfiguration (DFE, ADCE/AEQ, Eye Viewer). Use the EPE to estimate the incremental power consumption by these features.

9.2 Factors Affecting Power Consumption

Understanding the following factors that affect power consumption allows you to use the Power Analyzer and interpret its results effectively:

- Device Selection
- Environmental Conditions
- Device Resource Usage
- Signal Activities

9.2.1 Device Selection

Device families have different power characteristics. Many parameters affect the device family power consumption, including choice of process technology, supply voltage, electrical design, and device architecture.

(5) EPE and Power Analyzer outputs vary by device family. For more information, refer to the device-specific Early Power Estimators (EPE) and Power Analyzer Page and Power Analyzer Reports in the Intel Quartus Prime Help.

Power consumption also varies in a single device family. A larger device consumes more static power than a smaller device in the same family because of its larger transistor count. Dynamic power can also increase with device size in devices that employ global routing architectures.

The choice of device package also affects the ability of the device to dissipate heat. This choice can impact your required cooling solution choice to comply to junction temperature constraints.

Process variation can affect power consumption. Process variation primarily impacts static power because sub-threshold leakage current varies exponentially with changes in transistor threshold voltage. Therefore, you must consult device specifications for static power and not rely on empirical observation. Process variation has a weak effect on dynamic power.

9.2.2 Environmental Conditions

Operating temperature primarily affects device static power consumption. Higher junction temperatures result in higher static power consumption. The device thermal power and cooling solution that you use must result in the device junction temperature remaining within the maximum operating range for the device. The main environmental parameters affecting junction temperature are the cooling solution and ambient temperature.

The following table lists the environmental conditions that could affect power consumption.

Table 38. Environmental Conditions that Could Affect Power Consumption

Environmental Conditions	Description
Airflow	A measure of how quickly the device removes heated air from the vicinity of the device and replaces it with air at ambient temperature. You can either specify airflow as "still air" when you are not using a fan, or as the linear feet per minute rating of the fan in the system. Higher airflow decreases thermal resistance.
Heat Sink and Thermal Compound	A heat sink allows more efficient heat transfer from the device to the surrounding area because of its large surface area exposed to the air. The thermal compound that interfaces the heat sink to the device also influences the rate of heat dissipation. The case-to-ambient thermal resistance (θ_{CA}) parameter describes the cooling capacity of the heat sink and thermal compound employed at a given airflow. Larger heat sinks and more effective thermal compounds reduce θ_{CA}.
Junction Temperature	The junction temperature of a device is equal to: $T_{Junction} = T_{Ambient} + P_{Thermal} \cdot \theta_{JA}$ in which θ_{JA} is the total thermal resistance from the device transistors to the environment, having units of degrees Celsius per watt. The value θ_{JA} is equal to the sum of the junction-to-case (package) thermal resistance (θ_{JC}), and the case-to-ambient thermal resistance (θ_{CA}) of your cooling solution.
Board Thermal Model	The junction-to-board thermal resistance (θ_{JB}) is the thermal resistance of the path through the board, having units of degrees Celsius per watt. To compute junction temperature, you can use this board thermal model along with the board temperature, the top-of-chip θ_{JA} and ambient temperatures.

9.2.3 Device Resource Usage

The number and types of device resources used greatly affects power consumption.

- **Number, Type, and Loading of I/O Pins**—Output pins drive off-chip components, resulting in high-load capacitance that leads to a high-dynamic power per transition. Terminated I/O standards require external resistors that draw constant (static) power from the output pin.
- **Number and Type of Hard Logic Blocks**—A design with more logic elements (LEs), multiplier elements, memory blocks, transceiver blocks or HPS system tends to consume more power than a design with fewer circuit elements. The operating mode of each circuit element also affects its power consumption. For example, a DSP block performing 18 × 18 multiplications and a DSP block performing multiply-accumulate operations consume different amounts of dynamic power because of different amounts of charging internal capacitance on each transition. The operating mode of a circuit element also affects static power.
- **Number and Type of Global Signals**—Global signal networks span large portions of the device and have high capacitance, resulting in significant dynamic power consumption. The type of global signal is important as well. For example, Stratix V devices support global clocks and quadrant (regional) clocks. Global clocks cover the entire device, whereas quadrant clocks only span one-fourth of the device. Clock networks that span smaller regions have lower capacitance and tend to consume less power. The location of the logic array blocks (LABs) driven by the clock network can also have an impact because the Intel Quartus Prime software automatically disables unused branches of a clock.

9.2.4 Signal Activities

The behavior of each signal in your design is an important factor in estimating power consumption. The following table lists the two vital behaviors of a signal, which are toggle rate and static probability:

Table 39. Signal Behavior

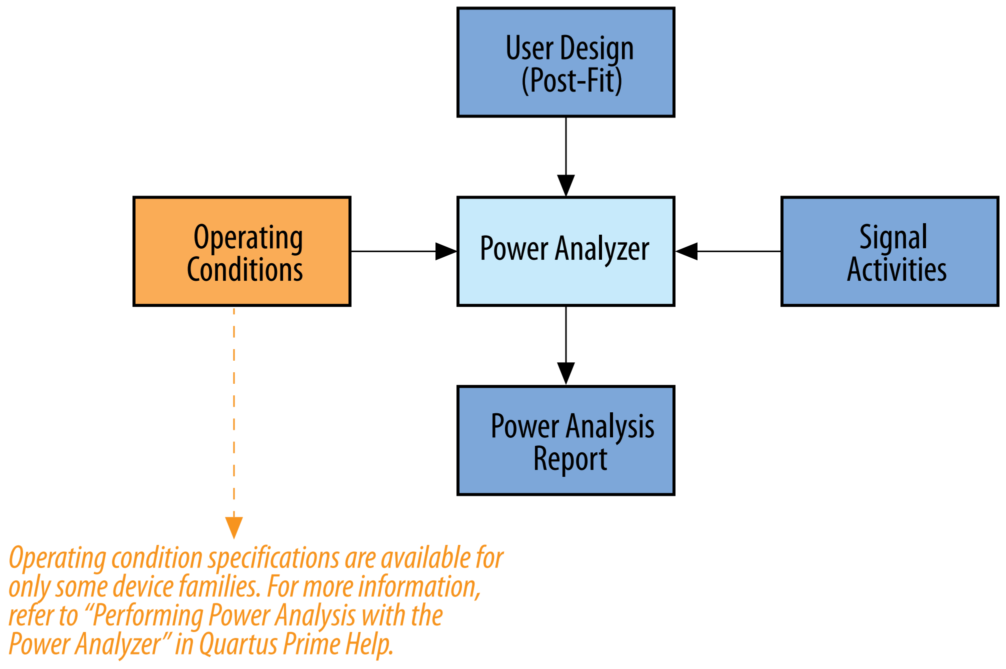
Signal Behavior	Description
Toggle rate	• The toggle rate of a signal is the average number of times that the signal changes value per unit of time. The units for toggle rate are transitions per second and a transition is a change from `1` to `0`, or `0` to `1`. • Dynamic power increases linearly with the toggle rate as you charge the board trace model more frequently for logic and routing. The Intel Quartus Prime software models full rail-to-rail switching. For high toggle rates, especially on circuit output I/O pins, the circuit can transition before fully charging the downstream capacitance. The result is a slightly conservative prediction of power by the Power Analyzer.
Static probability	• The static probability of a signal is the fraction of time that the signal is logic `1` during the period of device operation that is being analyzed. Static probability ranges from `0` (always at ground) to `1` (always at logic-high). • Static probabilities of their input signals can sometimes affect the static power that routing and logic consume. This effect is due to state-dependent leakage and has a larger effect on smaller process geometries. The Intel Quartus Prime software models this effect on devices at 90 nm or smaller if it is important to the power estimate. The static power also varies with the static probability of a logic `1` or `0` on the I/O pin when output I/O standards drive termination resistors.

Note: To get accurate results from the power analysis, the signal activities for analysis must represent the actual operating behavior of your design. Inaccurate signal toggle rate data is the largest source of power estimation error.

9.3 Power Analyzer Flow

The Power Analyzer supports accurate power estimations by allowing you to specify the important design factors affecting power consumption. The following figure shows the high-level Power Analyzer flow.

Figure 113. Power Analyzer High-Level Flow


User Design (Post-Fit)
Operating Conditions
Power Analyzer
Signal Activities
Power Analysis Report
Operating condition specifications are available for only some device families. For more information, refer to "Performing Power Analysis with the Power Analyzer" in Quartus Prime Help.

To obtain accurate I/O power estimates, the Power Analyzer requires you to synthesize your design and then fit your design to the target device. You must specify the electrical standard on each I/O cell and the board trace model on each I/O standard in your design.

9.3.1 Operating Settings and Conditions

You can specify device power characteristics, operating voltage conditions, and operating temperature conditions for power analysis in the Intel Quartus Prime software.

On the **Operating Settings and Conditions** page of the **Settings** dialog box, you can specify whether the device has typical power consumption characteristics or maximum power consumption characteristics.

On the **Voltage** page of the **Settings** dialog box, you can view the operating voltage conditions for each power rail in the device, and specify supply voltages for power rails with selectable supply voltages.

Note: The Intel Quartus Prime Fitter may override some of the supply voltages settings specified in this chapter. For example, supply voltages for some transceiver power supplies depend on the data rate used. If the Fitter detects that voltage required is different from the one specified in the **Voltage** page, it will automatically set the correct voltage for relevant rails. The Intel Quartus Prime Power Analyzer uses voltages selected by the Fitter if they conflict with the settings specified in the **Voltage** page.

On the **Temperature** page of the **Settings** dialog box, you can specify the thermal operating conditions of the device.

Related Links

- Operating Settings and Conditions Page (Settings Dialog Box)
- Voltage Page (Settings Dialog Box)
- Temperature Page (Settings Dialog Box)

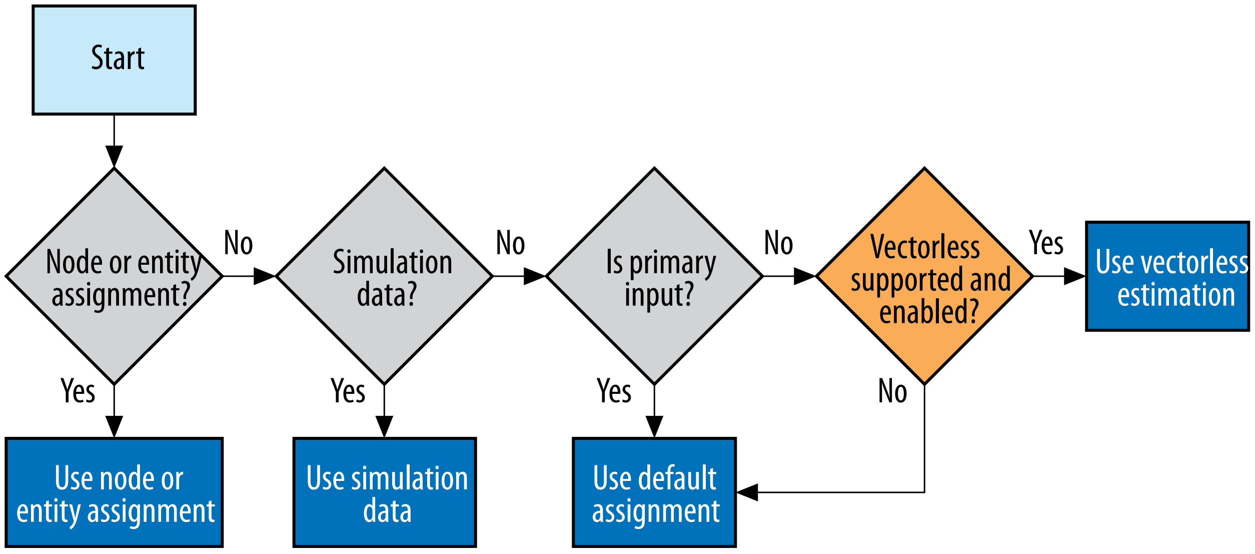
9.3.2 Signal Activities Data Sources

The Power Analyzer provides a flexible framework for specifying signal activities. The framework reflects the importance of using representative signal-activity data during power analysis. Use the following sources to provide information about signal activity:

- Simulation results
- User-entered node, entity, and clock assignments
- User-entered default toggle rate assignment
- Vectorless estimation

The Power Analyzer allows you to mix and match the signal-activity data sources on a signal-by-signal basis. The following figure shows the priority scheme applied to each signal.

Figure 114. Signal-Activity Data Source Priority Scheme


Start
Node or entity assignment?
No
Simulation data?
No
Is primary input?
No
Vectorless supported and enabled?
Yes
Use vectorless estimation
Yes
Use node or entity assignment
Yes
Use simulation data
Yes
Use default assignment
No

9.3.2.1 Simulation Results

The Power Analyzer directly reads the waveforms generated by a design simulation. Static probability and toggle rate can be calculated for each signal from the simulation waveform. Power analysis is most accurate when you use representative input stimuli to generate simulations.

The Power Analyzer reads results generated by the following simulators:

- ModelSim
- ModelSim - Intel FPGA Edition
- QuestaSim
- Active-HDL
- NCSim
- VCS
- VCS MX
- Riviera-PRO

Signal activity and static probability information are derived from a Verilog Value Change Dump File (`.vcd`). For more information, refer to Signal Activities on page 165.

For third-party simulators, use the EDA `Tool Settings` to specify the Generate Value Change Dump (VCD) file script option in the Simulation page of the Settings dialog box. These scripts instruct the third-party simulators to generate a `.vcd` that encodes the simulated waveforms. The Intel Quartus Prime Power Analyzer reads this file directly to derive the toggle rate and static probability data for each signal.

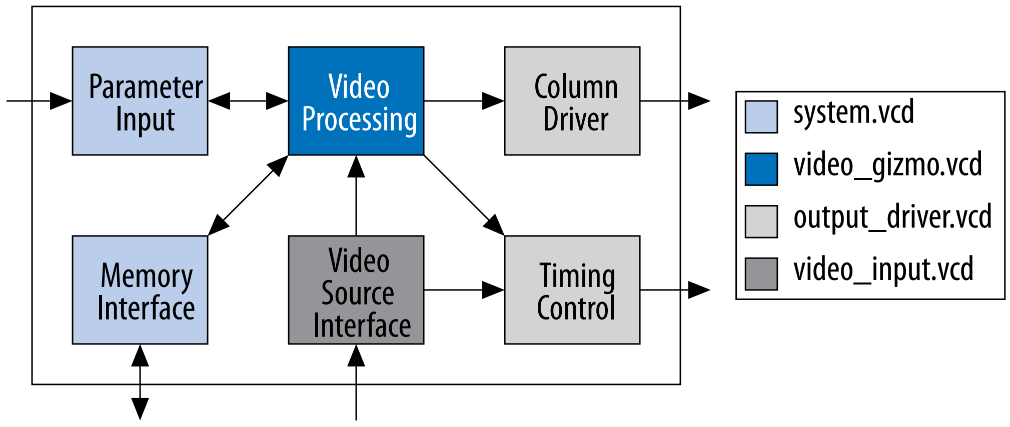
Third-party EDA simulators, other than those listed, can generate a `.vcd` that you can use with the Power Analyzer. For those simulators, you must manually create a simulation script to generate the appropriate `.vcd`.

Note: You can use a `.saf` created for power analysis to optimize your design for power during fitting by utilizing the appropriate settings in the power optimization list, available from **Assignments ➤ Settings ➤ Compiler Settings ➤ Advanced Settings (Fitter)**.

9.4 Using Simulation Files in Modular Design Flows

A common design practice is to create modular or hierarchical designs in which you develop each design entity separately, and then instantiate these modules in a higher-level entity to form a complete design. You can perform simulation on a complete design or on each module for verification. The Power Analyzer supports modular design flows when reading the signal activities from simulation files. The following figure shows an example of a modular design flow.

Figure 115. Modular Simulation Flow



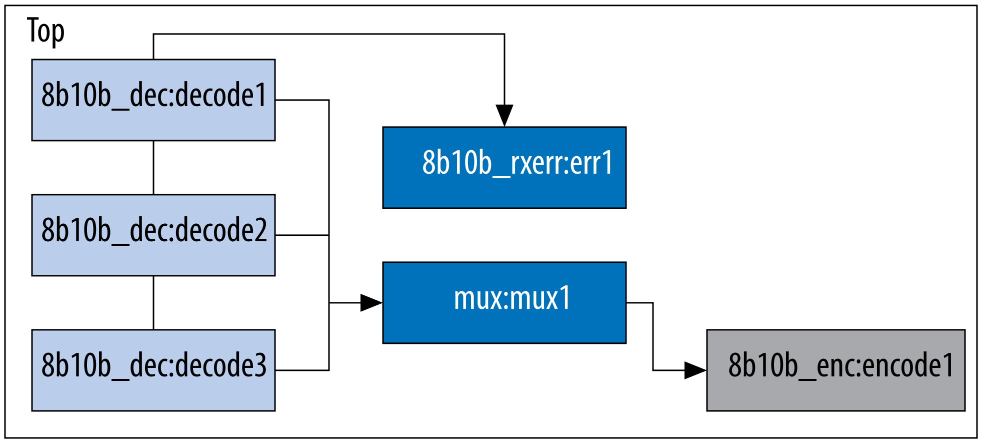
When specifying a simulation file (a `.vcd`), the software provides support to specify an associated design entity name, such that the Power Analyzer imports the signal activities derived from that file for the specified design entity. The Power Analyzer also supports the specification of multiple `.vcd` files for power analysis, with each having an associated design entity name to enable the integration of partial design simulations into a complete design power analysis. When specifying multiple `.vcd` files for your design, more than one simulation file can contain signal-activity information for the same signal.

Note: When you apply multiple `.vcd` files to the same design entity, the signal activity used in the power analysis is the equal-weight arithmetic average of each `.vcd`.

Note: When you apply multiple simulation files to design entities at different levels in your design hierarchy, the signal activity in the power analysis derives from the simulation file that applies to the most specific design entity.

The following figure shows an example of a hierarchical design. The top-level module of your design, called `Top`, consists of three 8b/10b decoders, followed by a `mux`. The software then encodes the output of the `mux` to produce the final output of the top-level module. An error-handling module handles any 8b/10b decoding errors. The Top module contains the top-level entity of your design and any logic not defined as part of another module. The design file for the top-level module might be a wrapper for the hierarchical entities below it, or it might contain its own logic. The following usage scenarios show common ways that you can simulate your design and import the `.vcd` into the Power Analyzer.

Figure 116. Example Hierarchical Design



9.4.1 Complete Design Simulation

You can simulate the entire design and generate a `.vcd` from a third-party simulator. The Power Analyzer can then import the `.vcd` (specifying the top-level design). The resulting power analysis uses the signal activities information from the generated `.vcd`, including those that apply to submodules, such as `decode [1-3]`, `err1`, `mux1`, and `encode1`.

9.4.2 Modular Design Simulation

You can independently simulate of the top-level design, and then import all the resulting `.vcd` files into the Power Analyzer. For example, you can simulate the `8b10b_dec` independent of the entire design and `mux`, `8b10b_rxerr`, and `8b10b_enc`. You can then import the `.vcd` files generated from each simulation by specifying the appropriate instance name. For example, if the files produced by the simulations are `8b10b_dec.vcd`, `8b10b_enc.vcd`, `8b10b_rxerr.vcd`, and `mux.vcd`, you can use the import specifications in the following table:

Table 40. Import Specifications

File Name	Entity
`8b10b_dec.vcd`	`Top|8b10b_dec:decode1`
`8b10b_dec.vcd`	`Top|8b10b_dec:decode2`
`8b10b_dec.vcd`	`Top|8b10b_dec:decode3`
`8b10b_rxerr.vcd`	`Top|8b10b_rxerr:err1`
`8b10b_enc.vcd`	`Top|8b10b_enc:encode1`
`mux.vcd`	`Top|mux:mux1`

The resulting power analysis applies the simulation vectors in each file to the assigned entity. Simulation provides signal activities for the pins and for the outputs of functional blocks. If the inputs to an entity instance are input pins for the entire design, the simulation file associated with that instance does not provide signal activities for the inputs of that instance. For example, an input to an entity such as `mux1` has its signal activity specified at the output of one of the decode entities.

9.4.3 Multiple Simulations on the Same Entity

You can perform multiple simulations of an entire design or specific modules of a design. For example, in the process of verifying the top-level design, you can have three different simulation testbenches: one for normal operation, and two for corner cases. Each of these simulations produces a separate `.vcd`. In this case, apply the different `.vcd` file names to the same top-level entity, as shown in the following table.

Table 41. Multiple Simulation File Names and Entities

File Name	Entity
`normal.vcd`	`Top`
`corner1.vcd`	`Top`
`corner2.vcd`	`Top`

The resulting power analysis uses an arithmetic average of the signal activities calculated from each simulation file to obtain the final signal activities used. If a signal `err_out` has a toggle rate of zero transition per second in `normal.vcd`, 50 transitions per second in `corner1.vcd`, and 70 transitions per second in `corner2.vcd`, the final toggle rate in the power analysis is 40 transitions per second.

If you do not want the Power Analyzer to read information from multiple instances and take an arithmetic average of the signal activities, use a `.vcd` that includes only signals from the instance that you care about.

9.4.4 Overlapping Simulations

You can perform a simulation on the entire design, and more exhaustive simulations on a submodule, such as `8b10b_rxerr`. The following table lists the import specification for overlapping simulations.

Table 42. Overlapping Simulation Import Specifications

File Name	Entity
`full_design.vcd`	`Top`
`error_cases.vcd`	`Top|8b10b_rxerr:err1`

In this case, the software uses signal activities from `error_cases.vcd` for all the nodes in the generated `.vcd` and uses signal activities from `full_design.vcd` for only those nodes that do not overlap with nodes in `error_cases.vcd`. In general, the more specific hierarchy (the most bottom-level module) derives signal activities for overlapping nodes.

9.4.5 Partial Simulations

You can perform a simulation in which the entire simulation time is not applicable to signal-activity calculation. For example, if you run a simulation for 10,000 clock cycles and reset the chip for the first 2,000 clock cycles. If the Power Analyzer performs the signal-activity calculation over all 10,000 cycles, the toggle rates are only 80% of their steady state value (because the chip is in reset for the first 20% of the simulation). In this case, you must specify the useful parts of the `.vcd` for power analysis. The **Limit VCD Period** option enables you to specify a start and end time when performing signal-activity calculations.

9.4.5.1 Specifying Start and End Time when Performing Signal-Activity Calculations using the Limit VCD Period Option

To specify a start and end time when performing signal-activity calculations using the **Limit VCD period** option, follow these steps:

1. In the Intel Quartus Prime software, on the Assignments menu, click **Settings**.
2. Under the Category list, click **Power Analyzer Settings**.
3. Turn on the **Use input file(s) to initialize toggle rates and static probabilities during power analysis** option.
4. Click **Add**.
5. In the **File name** and **Entity** fields, browse to the necessary files.
6. Under **Simulation period**, turn on **VCD file** and **Limit VCD period** options.
7. In the **Start time** and **End time** fields, specify the desired start and end time.
8. Click **OK**.

You can also use the following tcl or qsf assignment to specify `.vcd` files:

```
set_global_assignment -name POWER_INPUT_FILE_NAME "test.vcd" -section_id test.vcd
set_global_assignment -name POWER_INPUT_FILE_TYPE VCD -section_id test.vcd
set_global_assignment -name POWER_VCD_FILE_START_TIME "10 ns" -section_id test.vcd
set_global_assignment -name POWER_VCD_FILE_END_TIME "1000 ns" -section_id test.vcd
set_instance_assignment -name POWER_READ_INPUT_FILE test.vcd -to test_design
```

Related Links

- set_power_file_assignment
- Add/Edit Power Input File Dialog Box

9.4.6 Node Name Matching Considerations

Node name mismatches happen when you have `.vcd` applied to entities other than the top-level entity. In a modular design flow, the gate-level simulation files created in different Intel Quartus Prime projects might not match their node names with the current Intel Quartus Prime project.

For example, you may have a file named `8b10b_enc.vcd`, which the Intel Quartus Prime software generates in a separate project called `8b10b_enc` while simulating the `8b10b` encoder. If you import the `.vcd` into another project called `Top`, you might encounter name mismatches when applying the `.vcd` to the `8b10b_enc` module in the `Top` project. This mismatch happens because the Intel Quartus Prime software might name all the combinational nodes in the `8b10b_enc.vcd` differently than in the `Top` project.

You can avoid name mismatching with only RTL simulation data, in which register names do not change, or with an incremental compilation flow that preserves node names along with a gate-level simulation.

Note: To ensure accuracy, Intel FPGA recommends that you use an incremental compilation flow to preserve the node names of your design.

9.4.7 Glitch Filtering

The Power Analyzer defines a glitch as two signal transitions so closely spaced in time that the pulse, or glitch, occurs faster than the logic and routing circuitry can respond. The output of a transport delay model simulator contains glitches for some signals. The logic and routing structures of the device form a low-pass filter that filters out glitches that are tens to hundreds of picoseconds long, depending on the device family.

Some third-party simulators use different models than the transport delay model as the default model. Different models cause differences in signal activity and power estimation. The inertial delay model, which is the ModelSim default model, filters out more glitches than the transport delay model and usually yields a lower power estimate.

Note: Intel FPGA recommends that you use the transport simulation model when using the Intel Quartus Prime software glitch filtering support with third-party simulators. Simulation glitch filtering has little effect if you use the inertial simulation model.

Glitch filtering in a simulator can also filter a glitch on one logic element (LE) (or other circuit element) output from propagating to downstream circuit elements to ensure that the glitch does not affect simulated results. Glitch filtering prevents a glitch on one signal from producing non-physical glitches on all downstream logic, which can result in a signal toggle rate and a power estimate that are too high. Circuit elements in which every input transition produces an output transition, including multipliers and logic cells configured to implement XOR functions, are especially prone to glitches. Therefore, circuits with such functions can have power estimates that are too high when glitch filtering is not used.

Note: Intel FPGA recommends that you use the glitch filtering feature to obtain the most accurate power estimates. For `.vcd` files, the Power Analyzer flows support two levels of glitch filtering.

9.4.7.1 Enabling Tool Based Glitch Filtering

To enable the first level of glitch filtering in the Intel Quartus Prime software for supported third-party simulators, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Simulation under EDA Tool Settings**.
3. Select the **Tool name** to use for the simulation.
4. Turn on **Enable glitch filtering**.

9.4.7.2 Enabling Glitch Filtering During Power Analysis

The second level of glitch filtering occurs while the Power Analyzer is reading the .vcd generated by a third-party simulator. To enable the second level of glitch filtering, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Power Analyzer Settings**.
3. Under **Input File(s)**, turn on **Perform glitch filtering on VCD files**.

The .vcd file reader performs filtering complementary to the filtering performed during simulation and is often not as effective. While the .vcd file reader can remove glitches on logic blocks, the file reader cannot determine how a given glitch affects downstream logic and routing, and may eliminate the impact of the glitch completely. Filtering the glitches during simulation avoids switching downstream routing and logic automatically.

Note: When running simulation for design verification (rather than to produce input to the Power Analyzer), Intel recommends that you turn off the glitch filtering option to produce the most rigorous and conservative simulation from a functionality viewpoint. When performing simulation to produce input for the Power Analyzer, Intel FPGA recommends that you turn on the glitch filtering to produce the most accurate power estimates.

9.4.8 Node and Entity Assignments

You can assign toggle rates and static probabilities to individual nodes and entities in the design. These assignments have the highest priority, overriding data from all other signal-activity sources.

You must use the Assignment Editor or Tcl commands to create the **Power Toggle Rate** and **Power Static Probability** assignments. You can specify the power toggle rate as an absolute toggle rate in transitions per second using the **Power Toggle Rate** assignment, or you can use the **Power Toggle Rate Percentage** assignment to specify a toggle rate relative to the clock domain of the assigned node for a more specific assignment made in terms of hierarchy level.

Note: If you use the **Power Toggle Rate Percentage** assignment, and the node does not have a clock domain, the Intel Quartus Prime software issues a warning and ignores the assignment.

Assigning toggle rates and static probabilities to individual nodes and entities is appropriate for signals in which you have knowledge of the signal or entity being analyzed. For example, if you know that a 100 MHz data bus or memory output produces data that is essentially random (uncorrelated in time), you can directly enter a 0.5 static probability and a toggle rate of 50 million transitions per second.

The Power Analyzer treats bidirectional I/O pins differently. The combinational input port and the output pad for a pin share the same name. However, those ports might not share the same signal activities. For reading signal-activity assignments, the Power Analyzer creates a distinct name <*node_name~output*> when configuring the bidirectional signal as an output and <*node_name~result*> when configuring the signal as an input. For example, if a design has a bidirectional pin named MYPIN, assignments for the combinational input use the name MYPIN~result, and the assignments for the output pad use the name MYPIN~output.

Note: When you create the logic assignment in the Assignment Editor, you cannot find the `MYPIN~result` and `MYPIN~output` node names in the Node Finder. Therefore, to create the logic assignment, you must manually enter the two differentiating node names to create the assignment for the input and output port of the bidirectional pin.

9.4.8.1 Timing Assignments to Clock Nodes

For clock nodes, the Power Analyzer uses timing requirements to derive the toggle rate when neither simulation data nor user-entered signal-activity data is available. f_{MAX} requirements specify full cycles per second, but each cycle represents a rising transition and a falling transition. For example, a clock f_{MAX} requirement of 100 MHz corresponds to 200 million transitions per second for the clock node.

9.4.9 Default Toggle Rate Assignment

You can specify a default toggle rate for primary inputs and other nodes in your design. The Power Analyzer uses the default toggle rate when no other method specifies the signal-activity data.

The Power Analyzer specifies the toggle rate in absolute terms (transitions per second), or as a fraction of the clock rate in effect for each node. The toggle rate for a clock derives from the timing settings for the clock. For example, if the Power Analyzer specifies a clock with an f_{MAX} constraint of 100 MHz and a default relative toggle rate of 20%, nodes in this clock domain transition in 20% of the clock periods, or 20 million transitions occur per second. In some cases, the Power Analyzer cannot determine the clock domain for a node because either the Power Analyzer cannot determine a clock domain for the node, or the clock domain is ambiguous. For example, the Power Analyzer may not be able to determine a clock domain for a node if the user did not specify sufficient timing assignments. In these cases, the Power Analyzer substitutes and reports a toggle rate of zero.

9.4.10 Vectorless Estimation

For some device families, the Power Analyzer automatically derives estimates for signal activity on nodes with no simulation or user-entered signal-activity data. Vectorless estimation statistically estimates the signal activity of a node based on the signal activities of nodes feeding that node, and on the actual logic function that the node implements. Vectorless estimation cannot derive signal activities for primary inputs. Vectorless estimation is accurate for combinational nodes, but not for registered nodes. Therefore, the Power Analyzer requires simulation data for at least the registered nodes and I/O nodes for accuracy.

The **Power Analyzer Settings** dialog box allows you to disable vectorless estimation. When turned on, vectorless estimation takes precedence over default toggle rates. Vectorless estimation does not override clock assignments.

To disable vectorless estimation, perform the following steps:

1. In the Intel Quartus Prime software, on the Assignments menu, click **Settings**.
2. In the Category list, select **Power Analyzer Settings**.
3. Turn off the **Use vectorless estimation** option.

9.5 Using the Power Analyzer

For flows that use the Power Analyzer, you must first synthesize your design, and then fit it to the target device. You must either provide timing assignments for all the clocks in your design, or use a simulation-based flow to generate activity data. You must specify the I/O standard on each device input and output and the board trace model on each output in your design.

9.5.1 Common Analysis Flows

You can use the analysis flows in this section with the Power Analyzer. However, vectorless activity estimation is only available for some device families.

9.5.1.1 Signal Activities from RTL (Functional) Simulation, Supplemented by Vectorless Estimation

In the functional simulation flow, simulation provides toggle rates and static probabilities for all pins and registers in your design. Vectorless estimation fills in the values for all the combinational nodes between pins and registers, giving good results. This flow usually provides a compilation time benefit when you use the third-party RTL simulator.

9.5.1.1.1 RTL Simulation Limitation

RTL simulation may not provide signal activities for all registers in the post-fitting netlist because synthesis loses some register names. For example, synthesis might automatically transform state machines and counters, thus changing the names of registers in those structures.

9.5.1.2 Signal Activities from Vectorless Estimation and User-Supplied Input Pin Activities

The vectorless estimation flow provides a low level of accuracy, because vectorless estimation for registers is not entirely accurate.

9.5.1.3 Signal Activities from User Defaults Only

The user defaults only flow provides the lowest degree of accuracy.

9.5.2 Using .vcd for Power Estimation

Use a `.vcd` generated by your simulation tool as the source of activity data for accurate power estimation. The simulation `.vcd` includes all the routing resources and the exact logic array resource usage. Follow the documentation for your simulation tool to generate a `.vcd` during simulation. Specify the `.vcd` as the input to the Power Analyzer to estimate power for your design.

9.5.2.1 Generating a .vcd

In previous versions of the Intel Quartus Prime software, you could use either the Intel Quartus Prime simulator or an EDA simulator to perform your simulation. The Intel Quartus Prime software no longer supports a built-in simulator, and you must use an EDA simulator to perform simulation. Use the `.vcd` as the input to the Power Analyzer to estimate power for your design.

To create a `.vcd` for your design, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, under **EDA Tool Settings**, click **Simulation**.
3. In the **Tool name** list, select your preferred EDA simulator.
4. In the **Format for output netlist** list, select **Verilog HDL**, or **SystemVerilog HDL**, or **VHDL**.
5. Turn on **Generate Value Change Dump (VCD) file script**.
   This option turns on the **Map illegal HDL characters** and **Enable glitch filtering** options. The **Map illegal HDL characters** option ensures that all signals have legal names and that signal toggle rates are available later in the Power Analyzer. The **Enable glitch filtering** option directs the EDA Netlist Writer to perform glitch filtering when generating VHDL Output Files, Verilog Output Files, and the corresponding Standard Delay Format Output Files for use with other EDA simulation tools. This option is available regardless of whether or not you want to generate `.vcd` scripts.

   Note: When performing simulation using ModelSim, the **+nospecify** option for the `vsim` command disables the **specify path delays and timing checks** option in ModelSim. By enabling glitch filtering on the **Simulation** page, the simulation models include specified path delays. Thus, ModelSim might fail to simulate a design if you enabled glitch filtering and specified the **+nospecify** option. Intel FPGA recommends that you remove the **+nospecify** option from the ModelSim `vsim` command to ensure accurate simulation for power estimation.

6. Click **Script Settings**. Select the signals that you want to output to the `.vcd`. With **All signals** selected, the generated script instructs the third-party simulator to write all connected output signals to the `.vcd`. With **All signals except combinational lcell outputs** selected, the generated script tells the third-party simulator to write all connected output signals to the `.vcd`, except logic cell combinational outputs.

   Note: The file can become extremely large if you write all output signals to the file because the file size depends on the number of output signals being monitored and the number of transitions that occur.

7. Click **OK**.
8. In the **Design instance name** box, type a name for your testbench.
9. Compile your design with the Intel Quartus Prime software and generate the necessary EDA netlist and script that instructs the third-party simulator to generate a`.vcd`.
10. Perform a simulation with the third-party EDA simulation tool. Call the generated script in the simulation tool before running the simulation. The simulation tool generates the `.vcd` and places it in the project directory.

9.5.2.1.1 Generating a .vcd from ModelSim Software

To generate a `.vcd` with the ModelSim software, follow these steps:

1. In the Intel Quartus Prime software, on the Assignments menu, click **Settings**.
2. In the **Category** list, under **EDA Tool Settings**, click **Simulation**.
3. In the **Tool name** list, select your preferred EDA simulator.
4. In the **Format for output netlist** list, select **Verilog HDL**, or **SystemVerilog HDL**, or **VHDL**.
5. Turn on **Generate Value Change Dump (VCD) file script**.
6. To generate the`.vcd`, perform a full compilation.
7. In the ModelSim software, compile the files necessary for simulation.
8. Load your design by clicking **Start Simulation** on the Tools menu, or use the `vsim` command.
9. Use the `.vcd` script created in 6 on page 178 using the following command: `source <design>_dump_all_vcd_nodes.tcl`
10. Run the simulation (for example, run 2000ns or run -all).
11. Quit the simulation using the `quit -sim` command, if required.
12. Exit the ModelSim software.
    If you do not exit the software, the ModelSim software might end the writing process of the **.vcd** improperly, resulting in a corrupt `.vcd`.

9.6 Power Analyzer Compilation Report

The following table list the items in the Compilation Report of the Power Analyzer section.

Section	Description
Summary	The Summary section of the report shows the estimated total thermal power consumption of your design. This includes dynamic, static, and I/O thermal power consumption. The I/O thermal power includes the total I/O power drawn from the V_{CCIO} and V_{CCPD} power supplies and the power drawn from V_{CCINT} in the I/O subsystem including I/O buffers and I/O registers. The report also includes a confidence metric that reflects the overall quality of the data sources for the signal activities. For example, a **Low** power estimation confidence value reflects that you have provided insufficient toggle rate data, or most of the signal-activity information used for power estimation is from default or vectorless estimation settings. For more information about the input data, refer to the Power Analyzer Confidence Metric report.
Settings	The Settings section of the report shows the Power Analyzer settings information of your design, including the default input toggle rates, operating conditions, and other relevant setting information.
Simulation Files Read	The Simulation Files Read section of the report lists the simulation output file that the `.vcd` used for power estimation. This section also includes the file ID, file type, entity, VCD start time, VCD end time, the unknown percentage, and the toggle percentage. The unknown percentage indicates the portion of the design module unused by the simulation vectors.
Operating Conditions Used	The Operating Conditions Used section of the report shows device characteristics, voltages, temperature, and cooling solution, if any, during the power estimation. This section also shows the entered junction temperature or auto-computed junction temperature during the power analysis.
Thermal Power Dissipated by Block	The Thermal Power Dissipated by Block section of the report shows estimated thermal dynamic power and thermal static power consumption categorized by atoms. This information provides you with estimated power consumption for each atom in your design. By default, this section does not contain any data, but you can turn on the report with the **Write power dissipation by block to report file** option on the **Power Analyzer Settings** page.

continued...

Section	Description
Thermal Power Dissipation by Block Type (Device Resource Type)	This Thermal Power Dissipation by Block Type (Device Resource Type) section of the report shows the estimated thermal dynamic power and thermal static power consumption categorized by block types. This information is further categorized by estimated dynamic and static power and provides an average toggle rate by block type. Thermal power is the power dissipated as heat from the FPGA device.
Thermal Power Dissipation by Hierarchy	This Thermal Power Dissipation by Hierarchy section of the report shows estimated thermal dynamic power and thermal static power consumption categorized by design hierarchy. This information is further categorized by the dynamic and static power that was used by the blocks and routing in that hierarchy. This information is useful when locating modules with high power consumption in your design.
Core Dynamic Thermal Power Dissipation by Clock Domain	The Core Dynamic Thermal Power Dissipation by Clock Domain section of the report shows the estimated total core dynamic power dissipation by each clock domain, which provides designs with estimated power consumption for each clock domain in the design. If the clock frequency for a domain is unspecified by a constraint, the clock frequency is listed as "unspecified." For all the combinational logic, the clock domain is listed as no clock with zero MHz.
Current Drawn from Voltage Supplies	The Current Drawn from Voltage Supplies section of the report lists the current drawn from each voltage supply. The V_{CCIO} and V_{CCPD} voltage supplies are further categorized by I/O bank and by voltage. This section also lists the minimum safe power supply size (current supply ability) for each supply voltage. Minimum current requirement can be higher than user mode current requirement in cases in which the supply has a specific power up current requirement that goes beyond user mode requirement, such as the V_{CCPD} power rail in Stratix III and Stratix IV devices, and the V_{CCIO} power rail in Stratix IV devices. The I/O thermal power dissipation on the summary page does not correlate directly to the power drawn from the V_{CCIO} and V_{CCPD} voltage supplies listed in this report. This is because the I/O thermal power dissipation value also includes portions of the V_{CCINT} power, such as the I/O element (IOE) registers, which are modeled as I/O power, but do not draw from the V_{CCIO} and V_{CCPD} supplies. The reported current drawn from the I/O Voltage Supplies (ICCIO and ICCPD) as reported in the Power Analyzer report includes any current drawn through the I/O into off-chip termination resistors. This can result in ICCIO and ICCPD values that are higher than the reported I/O thermal power, because this off-chip current dissipates as heat elsewhere and does not factor in the calculation of device temperature. Therefore, total I/O thermal power does not equal the sum of current drawn from each V_{CCIO} and V_{CCPD} supply multiplied by V_{CCIO} and V_{CCPD} voltage. For SoC devices or for Arria V SoC and Cyclone V SoC devices, there is no standalone ICC_AUX_SHARED current drawn information. The ICC_AUX_SHARED is reported together with ICC_AUX.
Confidence Metric Details	The Confidence Metric is defined in terms of the total weight of signal activity data sources for both combinational and registered signals. Each signal has two data sources allocated to it; a toggle rate source and a static probability source. The Confidence Metric Details section also indicates the quality of the signal toggle rate data to compute a power estimate. The confidence metric is low if the signal toggle rate data comes from poor predictors of real signal toggle rates in the device during an operation. Toggle rate data that comes from simulation, user-entered assignments on specific signals or entities are reliable. Toggle rate data from default toggle rates (for example, 12.5% of the clock period) or vectorless estimation are relatively inaccurate. This section gives an overall confidence rating in the toggle rate data, from low to high. This section also summarizes how many pins, registers, and combinational nodes obtained their toggle rates from each of simulation, user entry, vectorless estimation, or default toggle rate estimations. This detailed information helps you understand how to increase the confidence metric, letting you determine your own confidence in the toggle rate data.
Signal Activities	The Signal Activities section lists toggle rates and static probabilities assumed by power analysis for all signals with fan-out and pins. This section also lists the signal type (pin, registered, or combinational) and the data source for the toggle rate and static probability. By default, this section does not contain any data, but you can turn on the report with the **Write signal activities to report file** option on the **Power Analyzer Settings** page. Intel recommends that you keep the **Write signal activities to report file** option turned off for a large design because of the large number of signals present. You can use the Assignment Editor to specify that activities for individual nodes or entities are reported by assigning an on value to those nodes for the **Power Report Signal Activities** assignment.
Messages	The Messages section lists the messages that the Intel Quartus Prime software generates during the analysis.

9.7 Scripting Support

You can run procedures and create settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For more information about scripting command options, refer to the Intel Quartus Prime Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ←
```

Related Links

- Tcl Scripting
- API Functions for Tcl
- Intel Quartus Prime Settings File Reference Manual

9.7.1 Running the Power Analyzer from the Command–Line

The executable to run the Power Analyzer is `quartus_pow`. For a complete listing of all command–line options supported by `quartus_pow`, type the following command at a system command prompt:

```
quartus_pow --help ←
```

or-

```
quartus_sh --qhelp ←
```

The following lists the examples of using the `quartus_pow` executable. Type the command listed in the following section at a system command prompt. These examples assume that operations are performed on Intel Quartus Prime project called *sample*.

To instruct the Power Analyzer to generate a EPE File:

```
quartus_pow sample --output_epe=sample.csv ←
```

To instruct the Power Analyzer to generate a EPE File without performing the power estimate:

```
quartus_pow sample --output_epe=sample.csv --estimate_power=off  ←
```

To instruct the Power Analyzer to use a .vcd as input (sample.vcd):

```
quartus_pow sample --input_vcd=sample.vcd ←
```

To instruct the Power Analyzer to use two .vcd files as input files (sample1.vcd and sample2.vcd), perform glitch filtering on the .vcd and use a default input I/O toggle rate of 10,000 transitions per second:

```
quartus_pow sample --input_vcd=sample1.vcd --input_vcd=sample2.vcd \
--vcd_filter_glitches=on --\
default_input_io_toggle_rate=10000transitions/s ←
```

To instruct the Power Analyzer to not use an input file, a default input I/O toggle rate of 60%, no vectorless estimation, and a default toggle rate of 20% on all remaining signals:

```
quartus_pow sample --no_input_file --default_input_io_toggle_rate=60% \
--use_vectorless_estimation=off --default_toggle_rate=20% ←
```

Note: No command–line options are available to specify the information found on the **Power Analyzer Settings Operating Conditions** page. Use the Intel Quartus Prime GUI to specify these options.

The `quartus_pow` executable creates a report file, <*revision name*>`.pow.rpt`. You can locate the report file in the main project directory. The report file contains the same information in Power Analyzer Compilation Report on page 178.

9.8 Document Revision History

The following table lists the revision history for this chapter.

Date	Version	Changes
2017.05.08	17.0.0	Removed references to PowerPlay® name. Power analysis occurs in the Intel Quartus Prime Power Analyzer.
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
2014.12.15	14.1.0	• Removed Signal Activities from Full Post-Fit Netlist (Timing) Simulation and Signal Activities from Full Post-Fit Netlist (Zero Delay) Simulation sections as these are no longer supported. • Updated location of Fitter Settings, Analysis & Synthesis Settings, and Physical Synthesis Optimizations to Compiler Settings.
2014.08.18	14.0a10.0	Updated "Current Drawn from Voltage Supplies" to clarify that for SoC devices or for Arria V SoC and Cyclone V SoC devices, there is no standalone ICC_AUX_SHARED current drawn information. The ICC_AUX_SHARED is reported together with ICC_AUX.
November 2012	12.1.0	• Updated "Types of Power Analyses" on page 8–2, and "Confidence Metric Details" on page 8–23. • Added "Importance of .vcd" on page 8–20, and "Avoiding Power Estimation and Hardware Measurement Mismatch" on page 8–24
June 2012	12.0.0	• Updated "Current Drawn from Voltage Supplies" on page 8–22. • Added "Using the HPS Power Calculator" on page 8–7.
November 2011	10.1.1	• Template update. • Minor editorial updates.
December 2010	10.1.0	• Added links to Intel Quartus Prime Help, removed redundant material. • Moved "Creating PowerPlay EPE Spreadsheets" to page 8–6. • Minor edits.
July 2010	10.0.0	• Removed references to the Intel Quartus Prime Simulator. • Updated Table 8–1 on page 8–6, Table 8–2 on page 8–13, and Table 8–3 on page 8–14. • Updated Figure 8–3 on page 8–9, Figure 8–4 on page 8–10, and Figure 8–5 on page 8–12.

continued...

Date	Version	Changes
November 2009	9.1.0	• Updated "Creating PowerPlay EPE Spreadsheets" on page 8–6 and "Simulation Results" on page 8–10. • Added "Signal Activities from Full Post-Fit Netlist (Zero Delay) Simulation" on page 8–19 and "Generating a .vcd from Full Post-Fit Netlist (Zero Delay) Simulation" on page 8–21. • Minor changes to "Generating a .vcd from ModelSim Software" on page 8–21. • Updated Figure 11–8 on page 11–24.
March 2009	9.0.0	• This chapter was chapter 11 in version 8.1. • Removed Figures 11-10, 11-11, 11-13, 11-14, and 11-17 from 8.1 version.
November 2008	8.1.0	• Updated for the Intel Quartus Prime software version 8.1. • Replaced Figure 11-3. • Replaced Figure 11-14.
May 2008	8.0.0	• Updated Figure 11–5. • Updated "Types of Power Analyses" on page 11–5. • Updated "Operating Conditions" on page 11–9. • Updated "PowerPlay Power Analyzer Compilation Report" on page 11–31. • Updated "Current Drawn from Voltage Supplies" on page 11–32.

Related Links

Documentation Archive
For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

10 System Debugging Tools Overview

The Intel FPGA system debugging tools help you verify your FPGA designs. As your product requirements continue to increase in complexity, the time you spend on design verification continues to rise. This chapter provides a quick overview of the tools available in the system debugging suite and discusses the criteria for selecting the best tool for your design.

10.1 System Debugging Tools Portfolio

The Intel Quartus Prime software provides a portfolio of system debugging tools for real-time verification of your design.

System debugging tools provide visibility by routing (or “tapping”) signals in your design to debugging logic. The Compiler includes the debugging logic in your design and generates programming files that you download into the FPGA or CPLD for analysis.

Each tool in the system debugging portfolio uses a combination of available memory, logic, and routing resources to assist in the debugging process. Because different designs have different constraints and requirements, you can choose the tool that matches the specific requirements for your design, such as the number of spare pins available or the amount of logic or memory resources remaining in the physical device.

10.1.1 System Debugging Tools Comparison

Table 43. Debugging Tools Portfolio

Tool	Description	Typical Usage
System Console	Provides real-time in-system debugging capabilities. Using System Console, you can read from and write to Memory Mapped components in our system without a processor or additional software. System Console uses a Tcl interpreter to communicate with hardware modules instantiated in your design. You can use it with the Transceiver Toolkit to monitor or debug your design. System Console uses Tcl as the fundamental infrastructure, so you can source scripts, set variables, write procedures, and take advantage of all the features of the Tcl scripting language.	You need to perform system-level debugging. For example, if you have an Avalon®-MM slave or Avalon-ST interfaces, you can debug your design at a transaction level. The tool supports JTAG connectivity and TCP/IP connectivity to the FPGA you want to debug.
Transceiver Toolkit	Allows you to test and tune transceiver link signal quality through a combination of metrics. Auto Sweeping of physical medium attachment (PMA) settings allows you to quickly find an optimal solution.	You need to debug or optimize signal integrity of your board layout even before the actual design to be run on the FPGA is ready.

continued...

Tool	Description	Typical Usage
Signal Tap Logic Analyzer	This logic analyzer uses FPGA resources to sample test nodes and outputs the information to the Intel Quartus Prime software for display and analysis.	You have spare on-chip memory and you want functional verification of your design running in hardware.
Signal Probe	This tool incrementally routes internal signals to I/O pins while preserving results from your last place-and-routed design.	You have spare I/O pins and you would like to check the operation of a small set of control pins using either an external logic analyzer or an oscilloscope.
Logic Analyzer Interface (LAI)	This tool multiplexes a larger set of signals to a smaller number of spare I/O pins. LAI allows you to select which signals are switched onto the I/O pins over a JTAG connection.	You have limited on-chip memory, and have a large set of internal data buses that you would like to verify using an external logic analyzer. Logic analyzer vendors, such as Tektronics and Agilent, provide integration with the tool to improve the usability of the tool.
In-System Sources and Probes	This utility provides an easy way to drive and sample logic values to and from internal nodes using the JTAG interface.	You want to prototype a front panel with virtual buttons for your FPGA design.
In-System Memory Content Editor	This tool displays and allows you to edit on-chip memory.	You would like to view and edit the contents of on-chip memory that is not connected to a Nios II processor. You can also use the tool when you do not want to have a Nios II debug core in your system.
Virtual JTAG Interface	This megafunction allows you to communicate with the JTAG interface so that you can develop your own custom applications.	You have custom signals in your design that you want to be able to communicate with.

10.1.2 System-Level Debugging Infrastructure

Intel FPGA on-chip debugging tools use the JTAG port to control and read-back data from debugging logic and signals under test. When your design includes multiple debugging blocks, all of the on-chip debugging tools share the JTAG resource.

Note: For System Console, you explicitly insert debug IP cores into your design to enable debugging.

During compilation, the Intel Quartus Prime software identifies the debugging blocks that use a JTAG interface and groups them under the System-Level Debugging Hub. This architecture allows you to instantiate multiple debugging tools in your design and use them simultaneously. The System-Level Debugging Hub appears in the design hierarchy of your project as `sld_hub:sld_hub_inst`.

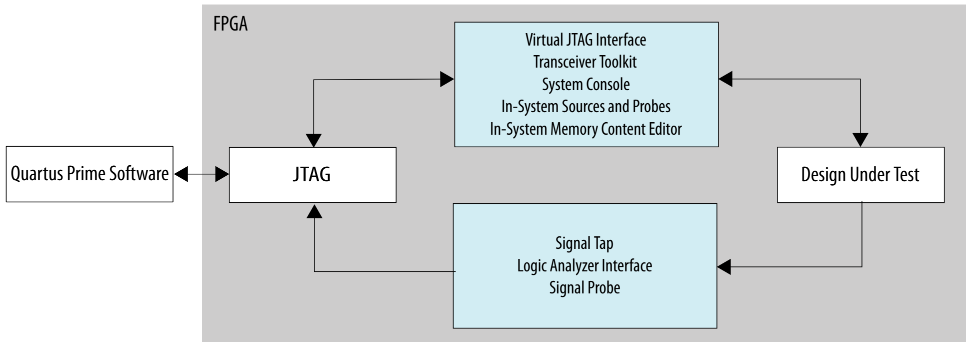
10.1.3 Debugging Ecosystem

The Intel Quartus Prime software allows you to use the debugging tools in tandem to exercise and analyze the logic under test and maximize closure. All debugging tools enable you to read back information gathered from the design nodes connected to the debugging logic.

Out of the set of debugging tools, the Signal Tap Logic Analyzer, the Logic Analyzer Interface, and the Signal Probe feature are general purpose troubleshooting tools optimized for probing signals in your register transfer level (RTL) netlist. In-System Sources and Probes, the Virtual JTAG Interface, System Console, Transceiver Toolkit, and In-System Memory Content Editor, allow you to read back data from breakpoints that you define, and to input values into your design during runtime.

Taken together, the set of on-chip debugging tools form a debugging ecosystem. The set of tools can generate a stimulus to and solicit a response from the logic under test, providing a complete solution.

Figure 117. Debugging Ecosystem at Runtime



10.1.4 Tools to Analyze RTL Nodes

The Signal Tap Logic Analyzer, Signal Probe, and LAI are designed specifically for probing and debugging RTL signals at system speed. These general-purpose analysis tools enable you to tap and analyze any routable node from the FPGA or CPLD.

- If you have spare logic and memory resources, the Signal Tap Logic Analyzer is useful for providing fast functional verification of your design running on actual hardware.

  Note: The Signal Tap Logic Analyzer is not supported on CPLDs, because there are no memory resources available on these devices.

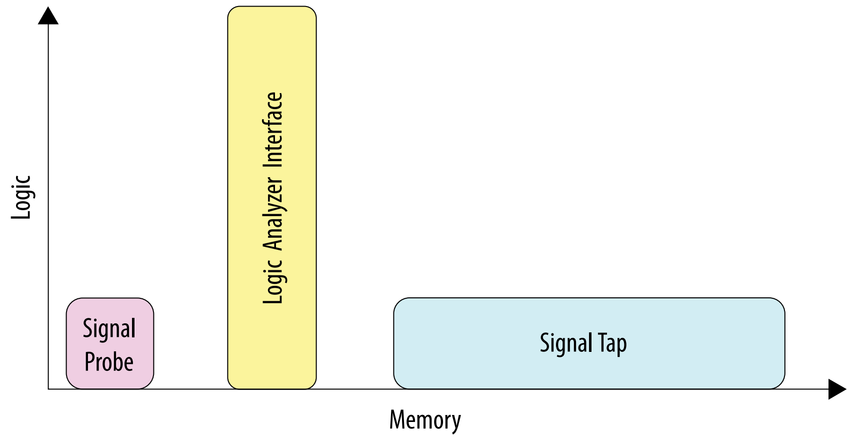
- Conversely, if logic and memory resources are tight and you require the large sample depths associated with external logic analyzers, both the LAI and the Signal Probe make it easy to view internal design signals using external equipment.

10.1.4.1 Resource Usage

The most important selection criteria for these three tools are the remaining resources on your device after implementing your design, and the number of spare pins.

Evaluate your debugging options early on in the design planning process to ensure that you support the appropriate options in your board, your Intel Quartus Prime project, and your design. Planning early can reduce time spent during debugging, and eliminates last minute changes to accommodate debug methodologies.

Figure 118. Resource Usage per Debugging Tool



10.1.4.1.1 Overhead Logic

Any debugging tool that requires a JTAG connection requires SLD infrastructure logic for communication with the JTAG interface and arbitration between instantiated debugging modules. This overhead logic uses around 200 logic elements (LEs), a small fraction of the resources available in any of the supported devices. All available debugging modules in your design share the overhead logic. Both the Signal Tap Logic Analyzer and the LAI use a JTAG connection.

For Signal Probe

Signal Probe requires very few on-chip resources. Because it requires no JTAG connection, Signal Probe uses no logic or memory resources. Signal Probe uses only routing resources to route an internal signal to a debugging test point.

For Logic Analyzer Interface

The LAI requires a small amount of logic to implement the multiplexing function between the signals under test, in addition to the SLD infrastructure logic. Because no data samples are stored on the chip, the LAI uses no memory resources.

For Signal Tap Logic Analyzer

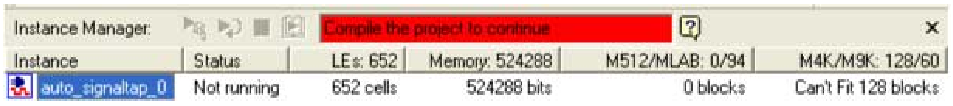
The Signal Tap Logic Analyzer requires both logic and memory resources. The number of logic resources used depends on the number of signals tapped and the complexity of the trigger logic. However, the amount of logic resources that the Signal Tap Logic Analyzer uses is typically a small percentage of most designs.

A baseline configuration consisting of the SLD arbitration logic and a single node with basic triggering logic contains approximately 300 to 400 Logic Elements (LEs). Each additional node you add to the baseline configuration adds about 11 LEs. Compared with logic resources, memory resources are a more important factor to consider for your design. Memory usage can be significant and depends on how you configure your Signal Tap Logic Analyzer instance to capture data and the sample depth that your design requires for debugging. For the Signal Tap Logic Analyzer, there is the added benefit of requiring no external equipment, as all of the triggering logic and storage is on the chip.

10.1.4.1.2 Resource Estimation

The resource estimation feature for the Signal Tap Logic Analyzer and the LAI allows you to quickly judge if enough on-chip resources are available before compiling the tool with your design.

Figure 119. Resource Estimator



Instance Manager: Compile the project to continue

Instance	Status	LEs: 652	Memory: 524288	M512/MLAB: 0/94	M4K/M9K: 128/60
auto_signaltap_0	Not running	652 cells	524288 bits	0 blocks	Can't Fit 128 blocks

10.1.4.2 Pin Usage

10.1.4.2.1 For Signal Probe

The ratio of the number of pins used to the number of signals tapped for the Signal Probe feature is one-to-one. Because this feature can consume free pins quickly, a typical application for this feature is routing control signals to spare pins for debugging.

10.1.4.2.2 For Logic Analyzer Interface

The ratio of the number of pins used to the number of signals tapped for the LAI is many-to-one. It can map up to 256 signals to each debugging pin, depending on available routing resources. The control of the active signals that are mapped to the spare I/O pins is performed via the JTAG port. The LAI is ideal for routing data buses to a set of test pins for analysis.

10.1.4.2.3 For Signal Tap Logic Analyzer

Other than the JTAG test pins, the Signal Tap Logic Analyzer uses no additional pins. All data is buffered using on-chip memory and communicated to the Signal Tap Logic Analyzer GUI via the JTAG test port.

10.1.4.3 Usability Enhancements

The Signal Tap Logic Analyzer, Signal Probe, and LAI tools can be added to your existing design with minimal effects. With the node finder, you can find signals to route to a debugging module without making any changes to your HDL files. Signal Probe inserts signals directly from your post-fit database. The Signal Tap Logic Analyzer and LAI support inserting signals from both pre-synthesis and post-fit netlists.

10.1.4.3.1 Incremental Compilation

All three tools allow you to find and configure your debugging setup quickly. In addition, the Intel Quartus Prime incremental compilation feature and the Intel Quartus Prime incremental routing feature allow for a fast turnaround time for your programming file, increasing productivity and enabling fast debugging closure.

Both the LAI and Signal Tap Logic Analyzer support incremental compilation. With incremental compilation, you can add a Signal Tap Logic Analyzer instance or an LAI instance incrementally into your placed-and-routed design. This has the benefit of both preserving your timing and area optimizations from your existing design, and decreasing the overall compilation time when any changes are necessary during the debugging process. With incremental compilation, you can save up to 70% compile time of a full compilation.

10.1.4.3.2 Incremental Routing

Signal Probe uses the incremental routing feature. The incremental routing feature runs only the Fitter stage of the compilation. This leaves your compiled design untouched, except for the newly routed node or nodes. With Signal Probe, you can save as much as 90% compile time of a full compilation.

10.1.4.3.3 Automation Via Scripting

As another productivity enhancement, all tools in the on-chip debugging tool set support scripting via the `quartus_stp` Tcl package. For the Signal Tap Logic Analyzer and the LAI, scripting enables user-defined automation for data collection while debugging in the lab. The System Console includes a full Tcl interpreter for scripting.

10.1.4.3.4 Remote Debugging

You can perform remote debugging of your system with the Intel Quartus Prime software via the System Console. This feature allows you to debug equipment deployed in the field through an existing TCP/IP connection.

There are two Application Notes available to assist you.

- Application Note 624 describes how to set up your Nios II system to use the System Console to perform remote debugging.
- Application Note 693 describes how to set up your Intel FPGA SoC to use the SLD tools to perform remote debugging.

Related Links

- Application Note 624: Debugging with System Console over TCP/IP
- Application Note 693: Remote Debugging over TCP/IP for Intel FPGA SoC

10.1.5 Suggested On-Chip Debugging Tools for Common Debugging Features

Table 44. Tools for Common Debugging Features [(1)]

Feature	Signal Probe	Logic Analyzer Interface (LAI)	Signal Tap Logic Analyzer	Description
Large Sample Depth	N/A	X	—	An external logic analyzer used with the LAI has a bigger buffer to store more captured data than the Signal Tap Logic Analyzer. No data is captured or stored with Signal Probe.
Ease in Debugging Timing Issue	X	X	—	External equipment, such as oscilloscopes and mixed signal oscilloscopes (MSOs), can be used with either LAI or Signal Probe. When used with the LAI, external equipment provides you with access to timing mode, which allows you to debug combined streams of data.
Minimal Effect on Logic Design	X	X [(2)]	X [(2)]	The LAI adds minimal logic to a design, requiring fewer device resources. The Signal Tap Logic Analyzer has little effect on the design, because it

continued...

Feature	Signal Probe	Logic Analyzer Interface (LAI)	Signal Tap Logic Analyzer	Description
				is set as a separate design partition. Signal Probe incrementally routes nodes to pins, not affecting the design at all.
Short Compile and Recompile Time	X	X [(2)]	X [(2)]	Signal Probe attaches incrementally routed signals to previously reserved pins, requiring very little recompilation time to make changes to source signal selections. The Signal Tap Logic Analyzer and the LAI can refit their own design partitions to decrease recompilation time.
Triggering Capability	N/A	N/A	X	The Signal Tap Logic Analyzer offers triggering capabilities that are comparable to commercial logic analyzers.
I/O Usage	—	—	X	No additional output pins are required with the Signal Tap Logic Analyzer. Both the LAI and Signal Probe require I/O pin assignments.
Acquisition Speed	N/A	—	X	The Signal Tap Logic Analyzer can acquire data at speeds of over 200 MHz. The same acquisition speeds are obtainable with an external logic analyzer used with the LAI, but might be limited by signal integrity issues.
No JTAG Connection Required	X	—	X	A FPGA design with the LAI requires an active JTAG connection to a host running the Intel Quartus Prime software. Signal Probe and Signal Tap do not require a host for debugging purposes.
No External Equipment Required	—	—	X	The Signal Tap Logic Analyzer logic is completely internal to the programmed FPGA device. No extra equipment is required other than a JTAG connection from a host running the Intel Quartus Prime software or the stand-alone Signal Tap Logic Analyzer software. Signal Probe and the LAI require the use of external debugging equipment, such as multimeters, oscilloscopes, or logic analyzers.

Notes to Table:
1. • X indicates the recommended tools for the feature.
   • — indicates that while the tool is available for that feature, that tool might not give the best results.
   • N/A indicates that the feature is not applicable for the selected tool.
2. When used with incremental compilation.

10.1.6 Stimulus-Capable Tools

The In-System Memory Content Editor, In-System Sources and Probes, and Virtual JTAG interface enable you to use the JTAG interface as a general-purpose communication port.

Though you can use all three tools to achieve the same results, there are some considerations that make one tool easier to use in certain applications. In-System Sources and Probes is ideal for toggling control signals. The In-System Memory Content Editor is useful for inputting large sets of test data. Finally, the Virtual JTAG interface is well suited for advanced users who want to develop their own customized JTAG solution.

System Console provides system-level debugging at a transaction level, such as with Avalon-MM slave or Avalon-ST interfaces. You can communicate to a chip through JTAG and TCP/IP protocols. System Console uses a Tcl interpreter to communicate with hardware modules that you instantiated into your design.

10.1.6.1 In-System Sources and Probes

In-System Sources and Probes is an easy way to access JTAG resources to both read and write to your design. You can start by instantiating a megafunction into your HDL code. The megafunction contains source ports and probe ports for driving values into and sampling values from the signals that are connected to the ports, respectively. Transaction details of the JTAG interface are abstracted away by the megafunction. During runtime, a GUI displays each source and probe port by instance and allows you to read from each probe port and drive to each source port. The GUI makes this tool ideal for toggling a set of control signals during the debugging process.

10.1.6.1.1 Push Button Functionality

A good application of In-System Sources and Probes is to use the GUI as a replacement for the push buttons and LEDs used during the development phase of a project. Furthermore, In-System Sources and Probes supports a set of scripting commands for reading and writing using `quartus_stp`. When used with the Tk toolkit, you can build your own graphical interfaces. This feature is ideal for building a virtual front panel during the prototyping phase of the design.

10.1.6.2 In-System Memory Content Editor

The In-System Memory Content Editor allows you to quickly view and modify memory content either through a GUI interface or through Tcl scripting commands. The In-System Memory Content Editor works by turning single-port RAM blocks into dual-port RAM blocks. One port is connected to your clock domain and data signals, and the other port is connected to the JTAG clock and data signals for editing or viewing.

10.1.6.2.1 Generate Test Vectors

Because you can modify a large set of data easily, a useful application for the In-System Memory Content Editor is to generate test vectors for your design. For example, you can instantiate a free memory block, connect the output ports to the logic under test (using the same clock as your logic under test on the system side), and create the glue logic for the address generation and control of the memory. At runtime, you can modify the contents of the memory using either a script or the In-System Memory Content Editor GUI and perform a burst transaction of the data contents in the modified RAM block synchronous to the logic being tested.

10.1.6.3 Virtual JTAG Interface Megafunction

The Virtual JTAG Interface megafunction provides the finest level of granularity for manipulating the JTAG resource. This megafunction allows you to build your own JTAG scan chain by exposing all of the JTAG control signals and configuring your JTAG Instruction Registers (IRs) and JTAG Data Registers (DRs). During runtime, you control the IR/DR chain through a Tcl API, or with System Console. This feature is meant for users who have a thorough understanding of the JTAG interface and want precise control over the number and type of resources used.

10.1.6.4 System Console

System Console is a framework that you can launch from the Intel Quartus Prime software to start services for performing various debugging tasks. System Console provides you with Tcl scripts and a GUI to access the Platform Designer (Standard) system integration tool to perform low-level hardware debugging of your design, as well as identify a module by its path, and open and close a connection to a Platform Designer (Standard) module. You can access your design at a system level for purposes of loading, unloading, and transferring designs to multiple devices. Also, System Console supports the Tk toolkit for building graphical interfaces.

10.1.6.4.1 Test Signal Integrity

System Console also allows you to access commands that allow you to control how you generate test patterns, as well as verify the accuracy of data generated by test patterns. You can use JTAG debug commands in System Console to verify the functionality and signal integrity of your JTAG chain. You can test clock and reset signals.

10.1.6.4.2 Board Bring-Up and Verification

You can use System Console to access programmable logic devices on your development board, perform board bring-up, and perform verification. You can also access software running on a Nios II or Intel FPGA SoC processor, as well as access modules that produce or consume a stream of bytes.

10.1.6.4.3 Test Link Signal Integrity with Transceiver Toolkit

Transceiver Toolkit runs from the System Console framework, and allows you to run automatic tests of your transceiver links for debugging and optimizing your transceiver designs. You can use the Transceiver Toolkit GUI to set up channel links in your transceiver devices and change parameters at runtime to measure signal integrity. For selected devices, the Transceiver Toolkit can also run and display eye contour tests.

10.2 Document Revision History

Table 45. Document Revision History

Date	Version	Changes
2017.05.08	17.0.0	• Combined Altera JTAG Interface and Required Arbitration Logic topics into a new updated topic named System-Level Debugging Infrastructure.
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
June 2014	14.0.0	Added information that System Console supports the Tk toolkit.
November 2013	13.1.0	Dita conversion. Added link to Remote Debugging over TCP/IP for Altera SoC Application Note.
June 2012	12.0.0	Maintenance release.

continued...

Date	Version	Changes
November 2011	10.0.2	Maintenance release. Changed to new document template.
December 2010	10.0.1	Maintenance release. Changed to new document template.
July 2010	10.0.0	Initial release

Related Links

Documentation Archive
For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

11 Analyzing and Debugging Designs with System Console

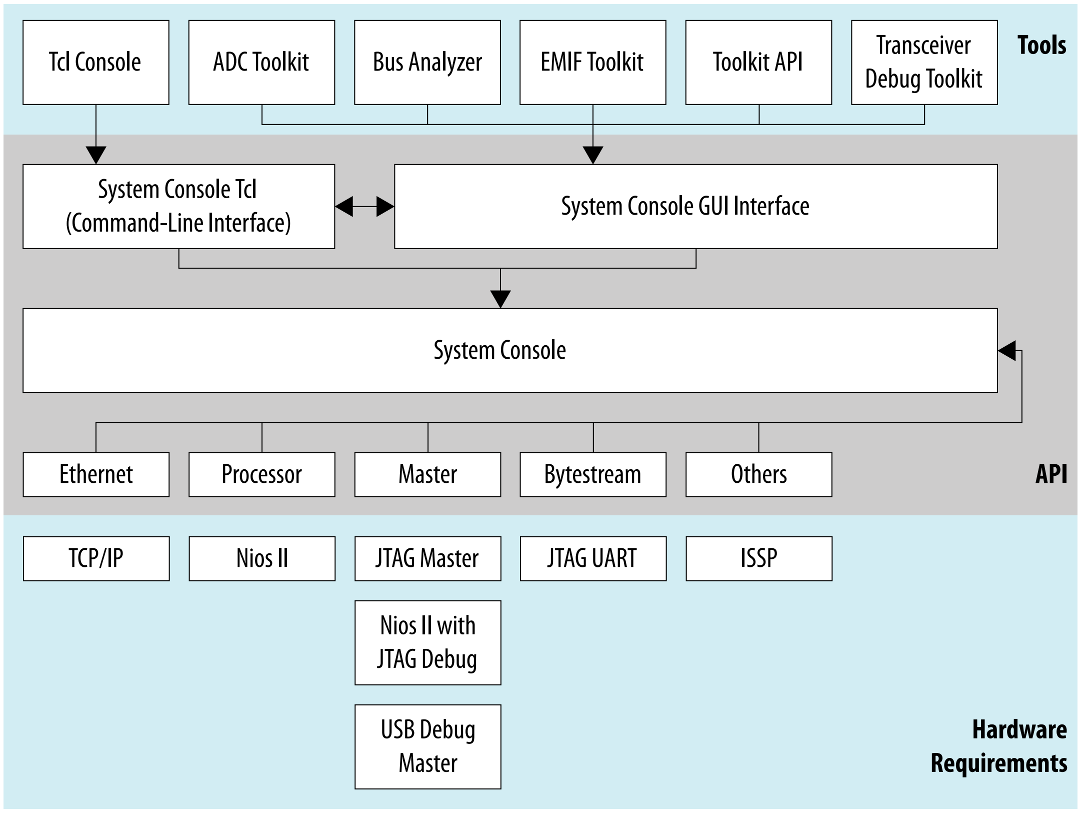
11.1 Introduction to System Console

System Console provides visibility into your design and allows you to perform system-level debug on a FPGA at run-time. System Console performs tests on debug-enabled Platform Designer (Standard) instantiated IP cores. A variety of debug services provide read and write access to elements in your design. You can perform the following tasks with System Console and the tools built on top of System Console:

- Bring up boards with both finalized and partially complete designs.
- Perform remote debug with internet access.
- Automate run-time verification through scripting across multiple devices in your system.
- Test serial links with point-and-click configuration tuning in the Transceiver Toolkit.
- Debug memory interfaces with the External Memory Interface Toolkit.
- Integrate your debug IP into the debug platform.
- Test the performance of your ADC and analog chain on a Intel MAX 10 device with the ADC Toolkit.
- Perform system verification with MATLAB/Simulink.

Figure 120. System Console Tools

(*Tools*) shows the applications that interact with System Console. The System Console API supports services that access your design in operation. Some services have specific hardware requirements.



Note: Use debug links to connect the host to the target you are debugging.

Related Links

- Introduction to Intel Memory Solution
  In *External Memory Interface Handbook Volume 1*
- Debugging Transceiver Links on page 274
- Application Note 693: Remote Hardware Debugging over TCP/IP for Intel SoC
- Application Note 624: Debugging with System Console over TCP/IP
- White Paper 01208: Hardware in the Loop from the MATLAB/Simulink Environment
- System Console Online Training

11.2 Debugging Flow with the System Console

To use the System Console you perform a series of steps:

1. Add an IP Core to your Platform Designer (Standard) system.
2. Generate your Platform Designer (Standard) system.
3. Compile your design.
4. Connect a board and program the FPGA.

5. Start System Console.
6. Locate and open a System Console service.
7. Perform debug operation(s) with the service.
8. Close the service.

11.3 IP Cores that Interact with System Console

System Console runs on your host computer and communicates with your running design through debug agents. Debug agents are soft-logic embedded in some IP cores that enable debug communication with the host computer.

You instantiate debug IP cores using the Platform Designer (Standard) IP Catalog. Some IP cores are enabled for debug by default, while you can enable debug for other IP cores through options in the parameter editor. Some debug agents have multiple purposes.

When you use IP cores with embedded debug in your design, you can make large portions of the design accessible. Debug agents allow you to read and write to memory and alter peripheral registers from the host computer.

Services associated with debug agents in the running design can open and close as needed. System Console determines the communication protocol with the debug agent. The communication protocol determines the best board connection to use for command and data transmission.

The Programmable SRAM Object File (`.sof`) provides the System Console with channel communication information. When System Console opens in the Intel Quartus Prime software or Platform Designer (Standard) while your design is open, any existing `.sof` is automatically found and linked to the detected running device. In a complex system, you may need to link the design and device manually.

Related Links

WP-01170 System-Level Debugging and Monitoring of FPGA Designs

11.3.1 Services Provided through Debug Agents

By adding the appropriate debug agent to your design, System Console services can use the associated capabilities of the debug agent.

Table 46. Common Services for System Console

Service	Function	Debug Agent Providing Service
master	Access memory-mapped (Avalon-MM or AXI) slaves connected to the master interface.	• Nios II with debug • JTAG to Avalon Master Bridge • USB Debug Master
slave	Allows the host to access a single slave without needing to know the location of the slave in the host's memory map. Any slave that is accessible to a System Console master can provide this service.	• Nios II with debug • JTAG to Avalon Master Bridge • USB Debug Master

continued...

Service	Function	Debug Agent Providing Service
		If an SRAM Object File (`.sof`) is loaded, then slaves controlled by a debug master provide the slave service.
processor	• Start, stop, or step the processor. • Read and write processor registers.	Nios II with debug
JTAG UART	The JTAG UART is an Avalon-MM slave device that you can use in conjunction with System Console to send and receive byte streams.	JTAG UART

Note: The following IP cores in the IP Catalog do not support VHDL simulation generation in the current version of the Intel Quartus Prime software:

- JTAG Debug Link
- SLD Hub Controller System
- USB Debug Link

Related Links

- System Console Examples and Tutorials on page 261
- System Console Commands on page 199

11.4 Starting System Console

11.4.1 Starting System Console from Nios II Command Shell

1. On the Windows Start menu, click **All Programs ➤ Intel ➤ Nios II EDS *<version>* ➤ Nios II*<version>* ➤ Command Shell.**.
2. Type `system-console`.
3. Type `-- help` for System Console help.
4. Type `system-console --project_dir=<project directory>` to point to a directory that contains `.qsf` or `.sof` files.

11.4.2 Starting Stand-Alone System Console

You can get the stand-alone version of System Console as part of the Intel Quartus Prime software Programmer and Tools installer on the Altera website.

1. Navigate to the **Download Center** page and click the **Additional Software** tab.
2. On the Windows Start menu, click **All Programs ➤ Intel FPGA *<version>* ➤ Programmer and Tools ➤ System Console**.

Related Links

Intel Download Center

11.4.3 Starting System Console from Platform Designer (Standard)

Click **Tools ➤ System Console**.

11.4.4 Starting System Console from Intel Quartus Prime

Click **Tools ➤ System Debugging Tools ➤ System Console**.

11.4.5 Customizing Startup

You can customize your System Console environment, as follows:

- Add commands to the `system_console_rc` configuration file located at:
  - `<$HOME>/system_console/system_console_rc.tcl`

  The file in this location is the user configuration file, which only affects the owner of the home directory.
- Specify your own design startup configuration file with the command-line argument `--rc_script=<path_to_script>`, when you launch System Console from the Nios II command shell.
- Use the `system_console_rc.tcl` file in combination with your custom `rc_script.tcl` file. In this case, the `system_console_rc.tcl` file performs System Console actions, and the `rc_script.tcl` file performs your debugging actions.

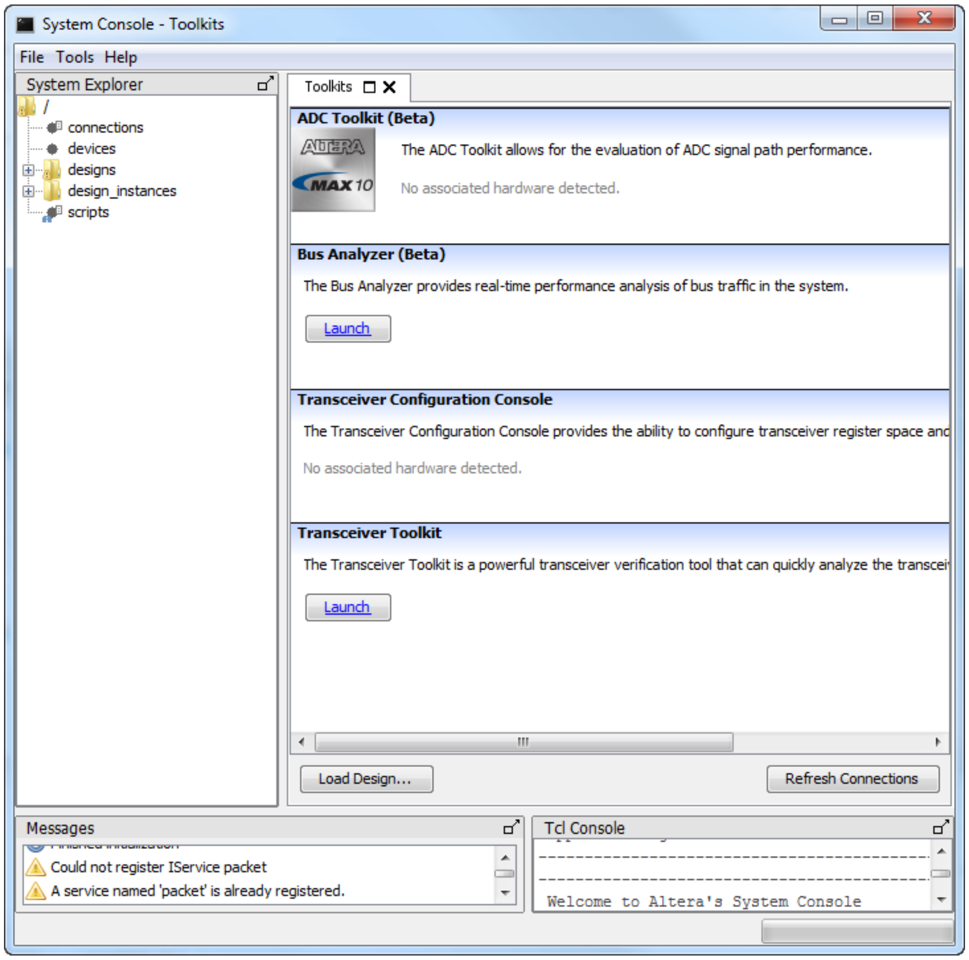
On startup, System Console automatically runs the Tcl commands in these files. The commands in the `system_console_rc.tcl` file run first, followed by the commands in the `rc_script.tcl` file.

11.5 System Console GUI

The System Console GUI consists of a main window with multiple panes, and allows you to interact with the design currently running on the host computer.

- **System Explorer**—Displays the hierarchy of the System Console virtual file system in your design, including board connections, devices, designs, and scripts.
- **Workspace**—Displays available toolkits including the ADC Toolkit, Transceiver Toolkit, Toolkits, GDB Server Control Panel, and Bus Analyzer. Click the **Tools** menu to launch applications.
- **Tcl Console**—A window that allows you to interact with your design using Tcl scripts, for example, sourcing scripts, writing procedures, and using System Console API.
- **Messages**—Displays status, warning, and error messages related to connections and debug actions.

Figure 121. System Console GUI



ADC Toolkit (Beta)

The ADC Toolkit allows for the evaluation of ADC signal path performance.

No associated hardware detected.

Bus Analyzer (Beta)

The Bus Analyzer provides real-time performance analysis of bus traffic in the system.

Transceiver Configuration Console

The Transceiver Configuration Console provides the ability to configure transceiver register space and

No associated hardware detected.

Transceiver Toolkit

The Transceiver Toolkit is a powerful transceiver verification tool that can quickly analyze the transcei

```
Could not register IService packet
A service named 'packet' is already registered.
```

```
Welcome to Altera's System Console
```

11.5.1 System Explorer Pane

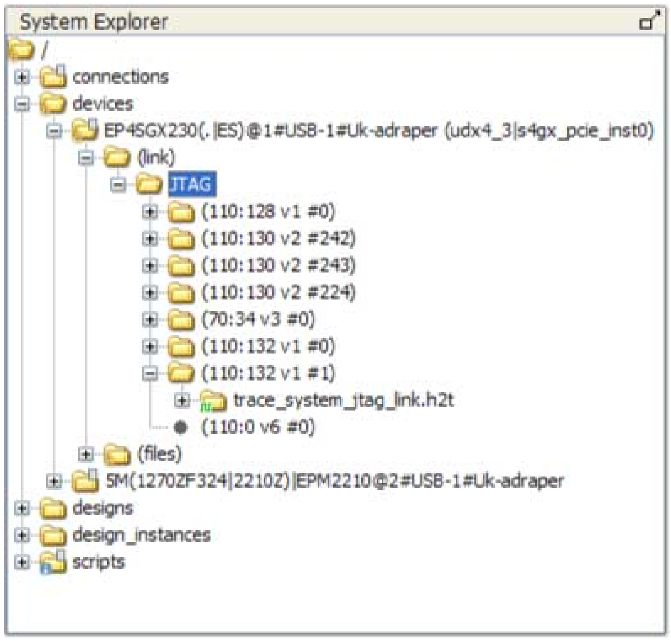
The **System Explorer** pane displays the virtual file system for all connected debugging IP cores, and contains the following information:

- **Devices** folder—Displays information about all devices connected to the System Console.
- **Scripts** folder—Stores scripts for easy execution.
- **Connections** folder—Displays information about the board connections visible to the System Console, such as Intel FPGA Download Cable. Multiple connections are possible.
- **Designs** folder—Displays information about Intel Quartus Prime designs connected to the System Console. Each design represents a loaded `.sof` file.

The **Devices** folder contains a sub-folder for each device connected to the System Console. Each device sub-folder contains a **(link)** folder, and may contain a **(files)** folder. The **(link)** folder shows debug agents (and other hardware) that System Console can access. The **(files)** folder contains information about the design files loaded from the Intel Quartus Prime project for the device.

Figure 122. System Explorer Pane

The figure shows the **EP4SGX230** folder under the **Device** folder, which contains a **(link)** folder. The **(link)** folder contains a **JTAG** folder, which describes the active debug connections to this device, for example, JTAG, USB, Ethernet, and agents connected to the EP4SGX230 device via a JTAG connection.



```
System Explorer
/
+ connections
- devices
  - EP4SGX230(.|ES)@1#USB-1#Uk-adraper (udx4_3|s4gx_pcie_inst0)
    - (link)
      - JTAG
        + (110:128 v1 #0)
        + (110:130 v2 #242)
        + (110:130 v2 #243)
        + (110:130 v2 #224)
        + (70:34 v3 #0)
        + (110:132 v1 #0)
        - (110:132 v1 #1)
          + trace_system_jtag_link.h2t
          ● (110:0 v6 #0)
    + (files)
  + 5M(1270ZF324|2210Z)|EPM2210@2#USB-1#Uk-adraper
+ designs
+ design_instances
+ scripts
```

- Folders with a context menu display a context menu icon. Right-click these folders to view the context menu. For example, the **Connections** folder above shows a context menu icon.
- Folders that have messages display a message icon. Mouse-over these folders to view the messages. For example, the **Scripts** folder in the example has a message icon.
- Debug agents that sense the clock and reset state of the target show an information or error message with a clock status icon. The icon indicates whether the clock is running (information, green), stopped (error, red), or running but in reset (error, red). For example, the **trace_system_jtag_link.h2t** folder in the figure has a running clock.

11.6 System Console Commands

The console commands enable testing. Use console commands to identify a service by its path, and to open and close the connection. The `path` that identifies a service is the first argument to most System Console commands.

To initiate a service connection, do the following:

1. Identify a service by specifying its path with the `get_service_paths` command.
2. Open a connection to the service with the `claim_service` command.
3. Use Tcl and System Console commands to test the connected device.
4. Close a connection to the service with the `close_service` command

Note: For all Tcl commands, the <*format*> argument must come first.

Table 47. System Console Commands

Command	Arguments	Function
`get_service_types`	N/A	Returns a list of service types that System Console manages. Examples of service types include master, bytestream, processor, sld, jtag_debug, device, and design.
`get_service_paths`	• <*service-type*> • <*device*>—Returns services in the same specified device. The argument can be a device or another service in the device. • <*hpath*>—Returns services whose `hpath` starts with the specified prefix. • <*type*>—Returns services whose debug type matches this value. Particularly useful when opening slave services. • <*type*>—Returns services on the same development boards as the argument. Specify a board service, or any other service on the same board.	Allows you to filter the services which are returned.
`claim_service`	• <*service-type*> • <*service-path*> • <*claim-group*> • <*claims*>	Provides finer control of the portion of a service you want to use. `claim_service` returns a new path which represents a use of that service. Each use is independent. Calling claim_service multiple times returns different values each time, but each allows access to the service until closed.
`close_service`	• <*service-type*> • <*service-path*>	Closes the specified service type at the specified path.
`is_service_open`	• <*service-type*> • <*service-type*>	Returns 1 if the service type provided by the path is open, 0 if the service type is closed.
`get_services_to_add`	N/A	Returns a list of all services that are instantiable with the `add_service` command.
`add_service`	• <*service-type*> • <*instance-name*> • *optional-parameters*	Adds a service of the specified service type with the given instance name. Run `get_services_to_add` to retrieve a list of instantiable services. This command returns the path where the service was added.

continued...

Command	Arguments	Function
		Run `help add_service` *<service-type>* to get specific help about that service type, including any parameters that might be required for that service.
`add_service gdbserver`	• *<Processor Service>* • *<port number>*	Instantiates a gdbserver.
`add_service tcp`	• *<instance name>* • *<ip_addr>* • *<port_number>*	Allows you to connect to a TCP/IP port that provides a debug link over ethernet. See AN693 (*Remote Hardware Debugging over TCP/IP for Intel FPGA SoC*) for more information.
`add_service transceiver_channel_rx`	• *<data_pattern_checker>* • *<path>* • *<transceiver path>* • *<transceiver channel address>* • *<reconfig path>* • *<reconfig channel address>*	Instantiates a Transceiver Toolkit receiver channel.
`add_service transceiver_channel_tx`	• *<data_pattern_generator >* • *<path>* • *<transceiver path>* • *<transceiver channel address>* • *<reconfig path>* • *<reconfig channel address>*	Instantiates a Transceiver Toolkit transmitter channel.
`add_service transceiver_debug_link`	• *<transceiver_channel_tx path>* • *<transceiver_channel_rx path>*	Instantiates a Transceiver Toolkit debug link.
`get_version`	N/A	Returns the current System Console version and build number.
`get_claimed_services`	• *<claim>*	For the given claim group, returns a list of services claimed. The returned list consists of pairs of paths and service types. Each pair is one claimed service.
`refresh_connections`	N/A	Scans for available hardware and updates the available service paths if there have been any changes.
`send_message`	• *<level>* • *<message>*	Sends a message of the given level to the message window. Available levels are info, warning, error, and debug.

Related Links

Remote Hardware Debugging over TCP/IP for SoC Devices

11.7 Running System Console in Command-Line Mode

You can run System Console in command line mode and either work interactively or run a Tcl script. System Console prints the output in the console window.

- `--cli`—Runs System Console in command-line mode.
- `--project_dir=<`*`project dir`*`>`—Directs System Console to the location of your hardware project. Also works in GUI mode.
- `--script=<`*`your script`*`>.tcl`—Directs System Console to run your Tcl script.
- `--help`— Lists all available commands. Typing `--help` <*command name*> provides the syntax and arguments of the command.

System Console provides command completion if you type the beginning letters of a command and then press the **Tab** key.

11.8 System Console Services

Intel's System Console services provide access to hardware modules instantiated in your FPGA. Services vary in the type of debug access they provide.

11.8.1 Locating Available Services

System Console uses a virtual file system to organize the available services, which is similar to the `/dev location` on Linux systems. Board connection, device type, and IP names are all part of a service path. Instances of services are referred to by their unique service path in the file system. To retrieve service paths for a particular service, use the command `get_service_paths` <*service-type*>.

Example 14. Locating a Service Path

```
#We are interested in master services.
set service_type "master"

#Get all the paths as a list.
set master_service_paths [get_service_paths $service_type]

#We are interested in the first service in the list.
set master_index 0

#The path of the first master.
set master_path [lindex $master_service_paths $master_index]

#Or condense the above statements into one statement:
set master_path [lindex [get_service_paths master] 0]
```

System Console commands require service paths to identify the service instance you want to access. The paths for different components can change between runs of System Console and between versions. Use the `get_service_paths` command to obtain service paths.

The string values of service paths change with different releases of the tool. Use the `marker_node_info` command to get information from the path.

System Console automatically discovers most services at startup. System Console automatically scans for all JTAG and USB-based service instances and retrieves their service paths. System Console does not automatically discover some services, such as TCP/IP. Use `add_service` to inform System Console about those services.

Example 15. Marker_node_info

Use the `marker_node_info` command to get information about the discovered services.

```
set slave_path [get_service_paths -type altera_avalon_uart.slave slave]
array set uart_info [marker_node_info $slave_path]
echo $uart_info(full_hpath)
```

11.8.2 Opening and Closing Services

After you have a service path to a particular service instance, you can access the service for use.

The `claim_service` command directs System Console to start using a particular service instance, and with no additional arguments, claims a service instance for exclusive use.

Example 16. Opening a Service

```
set service_type "master"
set claim_path [claim_service $service_type $master_path mylib];#Claims
service.
```

You can pass additional arguments to the `claim_service` command to direct System Console to start accessing a particular portion of a service instance. For example, if you use the master service to access memory, then use `claim_service` to only access the address space between `0x0` and `0x1000`. System Console then allows other users to access other memory ranges, and denies access to the claimed memory range. The `claim_service` command returns a newly created service path that you can use to access your claimed resources.

You can access a service after you open it. When you finish accessing a service instance, use the `close_service` command to direct System Console to make this resource available to other users.

Example 17. Closing a Service

```
close_service master $claim_path; #Closes the service.
```

11.8.3 SLD Service

The SLD Service shifts values into the instruction and data registers of SLD nodes and captures the previous value. When interacting with a SLD node, start by acquiring exclusive access to the node on an opened service.

Example 18. SLD Service

```
set timeout_in_ms 1000
set lock_failed [sld_lock $sld_service_path $timeout_in_ms]
```

This code attempts to lock the selected SLD node. If it is already locked, `sld_lock` waits for the specified timeout. Confirm the procedure returns non-zero before proceeding. Set the instruction register and capture the previous one:

```
if {$lock_failed} {
    return
}
```

```
set instr 7
set delay_us 1000
set capture [sld_access_ir $sld_service_path $instr $delay_us]
```

The 1000 microsecond delay guarantees that the following SLD command executes least 1000 microseconds later. Data register access works the same way.

```
set data_bit_length 32
set delay_us 1000
set data_bytes [list 0xEF 0xBE 0xAD 0xDE]
set capture [sld_access_dr $sld_service_path $data_bit_length $delay_us \
$data_bytes]
```

Shift count is specified in bits, but the data content is specified as a list of bytes. The capture return value is also a list of bytes. Always unlock the SLD node once finished with the SLD service.

```
sld_unlock $sld_service_path
```

Related Links

Virtual JTAG IP Core User Guide

11.8.3.1 SLD Commands

Table 48. SLD Commands

Command	Arguments	Function
`sld_access_ir`	*<claim-path>* *<ir-value>* *<delay>* (in µs)	Shifts the instruction value into the instruction register of the specified node. Returns the previous value of the instruction. If the *<delay>* parameter is non-zero, then the JTAG clock is paused for this length of time after the access.
`sld_access_dr`	*<service-path>* *<size_in_bits>* *<delay-in-µs>*, *<list_of_byte_values>*	Shifts the byte values into the data register of the SLD node up to the size in bits specified. If the *<delay>* parameter is non-zero, then the JTAG clock is paused for at least this length of time after the access. Returns the previous contents of the data register.
`sld_lock`	*<service-path>* *<timeout-in-milliseconds>*	Locks the SLD chain to guarantee exclusive access. Returns 0 if successful. If the SLD chain is already locked by another user, tries for *<timeout>*ms before throwing a Tcl error. You can use the catch command if you want to handle the error.
`sld_unlock`	*<service-path>*	Unlocks the SLD chain.

11.8.4 In-System Sources and Probes Service

The In-System Sources and Probes (ISSP) service provides scriptable access to the `altsource_probe` IP core in a similar manner to using the **In-System Sources and Probes Editor** in the Intel Quartus Prime software.

Example 19. ISSP Service

Before you use the ISSP service, ensure your design works in the **In-System Sources and Probes Editor**. In System Console, open the service for an ISSP instance.

```
set issp_index 0
set issp [lindex [get_service_paths issp] 0]
set claimed_issp [claim_service issp $issp mylib]
```

View information about this particular ISSP instance.

```
array set instance_info [issp_get_instance_info $claimed_issp]
set source_width $instance_info(source_width)
set probe_width $instance_info(probe_width)
```

The Intel Quartus Prime software reads probe data as a single bitstring of length equal to the probe width.

```
set all_probe_data [issp_read_probe_data $claimed_issp]
```

As an example, you can define the following procedure to extract an individual probe line's data.

```
proc get_probe_line_data {all_probe_data index} {
    set line_data [expr { ($all_probe_data >> $index) & 1 }]
    return $line_data
}
set initial_all_probe_data [issp_read_probe_data $claim_issp]
set initial_line_0 [get_probe_line_data $initial_all_probe_data 0]
set initial_line_5 [get_probe_line_data $initial_all_probe_data 5]
# ...
set final_all_probe_data [issp_read_probe_data $claimed_issp]
set final_line_0 [get_probe_line_data $final_all_probe_data 0]
```

Similarly, the Intel Quartus Prime software writes source data as a single bitstring of length equal to the source width.

```
set source_data 0xDEADBEEF
issp_write_source_data $claimed_issp $source_data
```

The currently set source data can also be retrieved.

```
set current_source_data [issp_read_source_data $claimed_issp]
```

As an example, you can invert the data for a 32-bit wide source by doing the following:

```
set current_source_data [issp_read_source_data $claimed_issp]
set inverted_source_data [expr { $current_source_data ^ 0xFFFFFFFF }]
issp_write_source_data $claimed_issp $inverted_source_data
```

11.8.4.1 In-System Sources and Probes Commands

Note: The valid values for ISSP claims include `read_only`, `normal`, and `exclusive`.

Table 49. In-System Sources and Probes Commands

Command	Arguments	Function

`issp_get_instance_info`	*<service-path>*	Returns a list of the configurations of the In-System Sources and Probes instance, including:
		`instance_index`
		`instance_name`
		`source_width`

continued...

Command	Arguments	Function
		`probe_width`
`issp_read_probe_data`	*<service-path>*	Retrieves the current value of the probe input. A hex string is returned representing the probe port value.
`issp_read_source_data`	*<service-path>*	Retrieves the current value of the source output port. A hex string is returned representing the source port value.
`issp_write_source_data`	*<service-path>* *<source-value>*	Sets values for the source output port. The value can be either a hex string or a decimal value supported by the System Console Tcl interpreter.

11.8.5 Monitor Service

The monitor service builds on top of the master service to allow reads of Avalon-MM slaves at a regular interval. The service is fully software-based. The monitor service requires no extra soft-logic. This service streamlines the logic to do interval reads, and it offers better performance than exercising the master service manually for the reads.

Example 20. Monitor Service

Start by determining a master and a memory address range that you are interested in polling continuously.

```
set master_index     0
set master [lindex [get_service_paths master] $master_index]
set address          0x2000
set bytes_to_read    100
set read_interval_ms 100
```

You can use the first master to read 100 bytes starting at address 0x2000 every 100 milliseconds. Open the monitor service:

```
set monitor [lindex [get_service_paths monitor] 0]
set claimed_monitor [claim_service monitor $monitor mylib]
```

Notice that the master service was not opened. The monitor service opens the master service automatically. Register the previously-defined address range and time interval with the monitor service:

```
monitor_add_range $claimed_monitor $master $address $bytes_to_read
monitor_set_interval $claimed_monitor $read_interval_ms
```

You can add more ranges. You must define the result at each interval:

```
global monitor_data_buffer
set monitor_data_buffer [list]
proc store_data {monitor master address bytes_to_read} {
  global monitor_data_buffer
  set data [monitor_read_data $claimed_monitor $master $address
$bytes_to_read]
  lappend monitor_data_buffer $data
}
```

The code example above, gathers the data and appends it with a global variable. `monitor_read_data` returns the range of data polled from the running design as a list. In this example, data will be a 100-element list. This list is then appended as a single element in the `monitor_data_buffer` global list. If this procedure takes longer than the interval period, the monitor service may have to skip the next one or more calls to the procedure. In this case, `monitor_read_data` will return the latest data polled. Register this callback with the opened monitor service:

```
set callback [list store_data $claimed_monitor $master $address
$bytes_to_read]
monitor_set_callback $claimed_monitor $callback
```

Use the callback variable to call when the monitor finishes an interval. Start monitoring:

```
monitor_set_enabled $claimed_monitor 1
```

Immediately, the monitor reads the specified ranges from the device and invokes the callback at the specified interval. Check the contents of `monitor_data_buffer` to verify this. To turn off the monitor, use 0 instead of 1 in the above command.

11.8.5.1 Monitor Commands

You can use the Monitor commands to read many Avalon-MM slave memory locations at a regular interval.

Under normal load, the monitor service reads the data after each interval and then calls the callback. If the value you read is timing sensitive, you can use the `monitor_get_read_interval` command to read the exact time between the intervals at which the data was read.

Under heavy load, or with a callback that takes a long time to execute, the monitor service skips some callbacks. If the registers you read do not have side effects (for example, they read the total number of events since reset), skipping callbacks has no effect on your code. The `monitor_read_data` command and `monitor_get_read_interval` command are adequate for this scenario.

If the registers you read have side effects (for example, they return the number of events since the last read), you must have access to the data that was read, but for which the callback was skipped. The `monitor_read_all_data` and `monitor_get_all_read_intervals` commands provide access to this data.

Table 50. Main Monitoring Commands

Command	Arguments	Function
`monitor_add_range`	*<service-path>* *<target-path>* *<address>* *<size>*	Adds a contiguous memory address into the monitored memory list. *<service path>* is the value returned when you opened the service. *<target-path>* argument is the name of a master service to read. The address is within the address space of this service. *<target-path>* is returned from `[lindex [get_service_paths master] `*n*`]` where *n* is the number of the master service. *<address>* and *<size>* are relative to the master service.
`monitor_set_callback`	*<service-path>* *<Tcl-expression>*	Defines a Tcl expression in a single string that will be evaluated after all the memories monitored by this service are read. Typically, this expression should be specified as a Tcl procedure call with necessary argument passed in.

continued...

Command	Arguments	Function
`monitor_set_interval`	*<service-path>* *<interval>*	Specifies the frequency of the polling action by specifying the interval between two memory reads. The actual polling frequency varies depending on the system activity. The monitor service will try to keep it as close to this specification as possible.
`monitor_get_interval`	*<service-path>*	Returns the current interval set which specifies the frequency of the polling action.
`monitor_set_enabled`	*<service-path>* *<enable(1)/ disable(0)>*	Enables and disables monitoring. Memory read starts after this is enabled, and Tcl callback is evaluated after data is read.

Table 51. Monitor Callback Commands

Command	Arguments	Function
`monitor_add_range`	*<service-path>* *<target-path>* *<address>* *<size>*	Adds contiguous memory addresses into the monitored memory list. The *<target-path>* argument is the name of a master service to read. The address is within the address space of this service.
`monitor_set_callback`	*<service-path>* *<Tcl-expression>*	Defines a Tcl expression in a single string that will be evaluated after all the memories monitored by this service are read. Typically, this expression should be specified as a Tcl procedure call with necessary argument passed in.
`monitor_read_data`	*<service-path>* *<target-path>* *<address>* *<size>*	Returns a list of 8-bit values read from the most recent values read from device. The memory range specified must be the same as the monitored memory range as defined by `monitor_add_range`.
`monitor_read_all_data`	*<service-path>* *<target-path>* *<address>* *<size>*	Returns a list of 8-bit values read from all recent values read from device since last Tcl callback. The memory range specified must be within the monitored memory range as defined by `monitor_add_range`.
`monitor_get_read_interval`	*<service-path>* *<target-path>* *<address>* *<size>*	Returns the number of milliseconds between last two data reads returned by `monitor_read_data`.
`monitor_get_all_read_intervals`	*<service-path>* *<target-path>* *<address>* *<size>*	Returns a list of intervals in milliseconds between two reads within the data returned by `monitor_read_all_data`.
`monitor_get_missing_event_count`	*<service-path>*	Returns the number of callback events missed during the evaluation of last Tcl callback expression.

11.8.6 Device Service

The device service supports device-level actions.

Example 21. Programming

You can use the device service with Tcl scripting to perform device programming.

```
set device_index 0 ; #Device index for target
set device [lindex [get_service_paths device] $device_index]
set sof_path [file join project_path output_files project_name.sof]
device_download_sof $device $sof_path
```

To program, all you need are the device service path and the file system path to a `.sof`. Ensure that no other service (e.g. master service) is open on the target device or else the command fails. Afterwards, you may do the following to check that the design linked to the device is the same one programmed:

```
device_get_design $device
```

11.8.6.1 Device Commands

The device commands provide access to programmable logic devices on your board. Before you use these commands, identify the path to the programmable logic device on your board using the `get_service_paths`.

Table 52. Device Commands

Command	Arguments	Function
`device_download_sof`	*<service_path>* *<sof-file-path>*	Loads the specified `.sof` to the device specified by the path.
`device_get_connections`	*<service_path>*	Returns all connections which go to the device at the specified path.
`device_get_design`	*<device_path>*	Returns the design this device is currently linked to.

11.8.7 Design Service

You can use design service commands to work with Intel Quartus Prime design information.

Example 22. Load

When you open System Console from the Intel Quartus Prime software or Platform Designer (Standard), the current project's debug information is sourced automatically if the `.sof` has been built. In other situations, you can load manually.

```
set sof_path [file join project_dir output_files project_name.sof]
set design [design_load $sof_path]
```

System Console is now aware that this particular `.sof` has been loaded.

Example 23. Linking

Once a `.sof` is loaded, System Console automatically links design information to the connected device. The resultant link persists and you can choose to unlink or reuse the link on an equivalent device with the same `.sof`.

You can perform manual linking.

```
set device_index 0; # Device index for our target
set device [lindex [get_service_paths device] $device_index]
design_link $design $device
```

Manually linking fails if the target device does not match the design service.

Linking fails even if the `.sof` programmed to the target is not the same as the design `.sof`.

11.8.7.1 Design Service Commands

Design service commands load and work with your design at a system level.

Table 53. Design Service Commands

Command	Arguments	Function
`design_load`	*<quartus-project-path>*, *<sof-file-path>*, or *<qpf-file-path>*	Loads a model of a Intel Quartus Prime design into System Console. Returns the design path. For example, if your Intel Quartus Prime Project File (`.qpf`) is in `c:/projects/loopback`, type the following command: `design_load {c:\projects\loopback\}`
`design_link`	*<design-path>* *<device-service-path>*	Links a Intel Quartus Prime logical design with a physical device. For example, you can link a Intel Quartus Prime design called **2c35_quartus_design** to a 2c35 device. After you create this link, System Console creates the appropriate correspondences between the logical and physical submodules of the Intel Quartus Prime project.
`design_extract_debug_files`	*<design-path>* *<zip-file-name>*	Extracts debug files from a `.sof` to a zip file which can be emailed to *Intel FPGA Support* for analysis. You can specify a design path of {} to unlink a device and to disable auto linking for that device.
`design_get_warnings`	*<design-path>*	Gets the list of warnings for this design. If the design loads correctly, then an empty list returns.

11.8.8 Bytestream Service

The bytestream service provides access to modules that produce or consume a stream of bytes. Use the bytestream service to communicate directly to the IP core that provides bytestream interfaces, such as the Altera JTAG UART or the Avalon-ST JTAG interface.

Example 24. Bytestream Service

The following code finds the bytestream service for your interface and opens it.

```
set bytestream_index 0
set bytestream [lindex [get_service_paths bytestream] $bytestream_index]
set claimed_bytestream [claim_service bytestream $bytestream mylib]
```

To specify the outgoing data as a list of bytes and send it through the opened service:

```
set payload [list 1 2 3 4 5 6 7 8]
bytestream_send $claimed_bytestream $payload
```

Incoming data also comes as a list of bytes.

```
set incoming_data [list]
while {[llength $incoming_data] ==0} {
    set incoming_data [bytestream_receive $claimed_bytestream 8]
}
```

Close the service when done.

```
close_service bytestream $claimed_bytestream
```

11.8.8.1 Bytestream Commands

Table 54. Bytestream Commands

Command	Arguments	Function
`bytestream_send`	*<service-path>* *<values>*	Sends the list of bytes to the specified bytestream service. Values argument is the list of bytes to send.
`bytestream_receive`	*<service-path>* *<length>*	Returns a list of bytes currently available in the specified services receive queue, up to the specified limit. Length argument is the maximum number of bytes to receive.

11.8.9 JTAG Debug Service

The JTAG Debug service allows you to check the state of clocks and resets within your design.

The following is a JTAG Debug design flow example.

1. To identify available JTAG Debug paths:

```
get_service_paths jtag_debug
```

2. To select a JTAG Debug path:

```
set jtag_debug_path [lindex [get_service_paths jtag_debug] 0]
```

3. To claim a JTAG Debug service path:

```
 set claim_jtag_path [claim_service jtag_debug$jtag_debug_path mylib]
```

4. Running the JTAG Debug service:

```
jtag_debug_reset_system $claim_jtag_path
jtag_debug_loop $claim_jtag_path [list 1 2 3 4 5]
```

11.8.9.1 JTAG Debug Commands

JTAG Debug commands help debug the JTAG Chain connected to a device.

Table 55. JTAG Debug Commands

Command	Argument	Function
`jtag_debug_loop`	*<service-path>* *<list_of_byte_values>*	Loops the specified list of bytes through a loopback of `tdi` and `tdo` of a system-level debug (SLD) node. Returns the list of byte values in the order that they were received. Blocks until all bytes are received. Byte values have the `0x` (hexadecimal) prefix and are delineated by spaces.
`jtag_debug_sample_clock`	*<service-path>*	Returns the value of the clock signal of the system clock that drives the module's system interface. The clock value is sampled asynchronously; consequently, you may need to sample the clock several times to guarantee that it is toggling.

continued...

Command	Argument	Function
`jtag_debug_sample_reset`	*<service-path>*	Returns the value of the `reset_n` signal of the Avalon-ST JTAG Interface core. If `reset_n` is low (asserted), the value is 0 and if `reset_n` is high (deasserted), the value is 1.
`jtag_debug_sense_clock`	*<service-path>*	Returns the result of a sticky bit that monitors for system clock activity. If the clock has toggled since the last execution of this command, the bit is 1. Returns `true` if the bit has ever toggled and otherwise returns `false`. The sticky bit is reset to 0 on read.
`jtag_debug_reset_system`	*<service-path>*	Issues a reset request to the specified service. Connectivity within your device determines which part of the system is reset.

11.9 Working with Toolkits

The Toolkit API allows you to create custom tools to visualize and interact with your design debug data. The Toolkit API provides graphical widgets in the form of buttons and text fields, which can leverage user input to interact with debug logic. You can use Toolkit API with the Intel Quartus Prime software versions 14.1 and later. The Toolkit API is the successor to the Dashboard service.

Toolkits you create with the Toolkit API require the following files:

- XML file that describes the toolkit (`.toolkit` file).
- Tcl file that implements the toolkit GUI.

11.9.1 Convert your Dashboard Scripts to Toolkit API

Convert your Dashboard scripts to work with the Toolkit API by following these steps:

1. Create a `.toolkit` file.
2. Modify your dashboard script:
   a. Remove the `add_service dashboard` *<name of service>* command.
   b. Change `dashboard_`*<command>* to `toolkit_`*<command>*.
   c. Change `open_service` to `claim_service`

   For example:

```
open_service slave $path
master_read_memory $path address count
```

   becomes

```
set c [claim_service slave $path lib {}]
master_read_memory $c address count
```

11.9.2 Creating a Toolkit Description File

A toolkit description file (`.toolkit`) is a XML file which provides the registration data for a toolkit.

Include the following attributes in your toolkit description file:

Table 56. Attributes in Toolkit Description File

Attribute name	Purpose
`name`	Internal toolkit file name.
`displayName`	Toolkit display name to appear in the GUI.
`addMenuItem`	Whether the System Console **Tools ➤ Toolkits** menu displays the toolkit.

Table 57. Toolkit child elements

Attribute name	Purpose
`description`	Description of the purpose of the toolkit.
`file`	Path to `.tcl` file containing the toolkit implementation.
`icon`	Path to icon to display as the toolkit launcher button in System Console *Note:* The `.png` 64x64 format is preferred. If the icon does not take up the whole space, ensure that the background is transparent.
`requirement`	If the toolkit works with a particular type of hardware, this attribute specifies the debug type name of the hardware. This attribute enables automatic discovery of the toolkit. The syntax of a toolkit's debug type name is: • Name of the `hw.tcl` component. • dot. • Name of the interface within that component which the toolkit uses. For example: `<hw.tcl name>.<interface name>`.

Example 25. .toolkit Description File

```
<?xml version="1.0" encoding="UTF-8"?>
        <toolkit name="toolkit_example" displayName="Toolkit Example"
addMenuItem="true">
        <file> toolkit_example.tcl </file>
        </toolkit>
```

Related Links

Matching Toolkits with IP Cores on page 214

11.9.3 Registering a Toolkit

Use the `toolkit_register` command in the System Console to make your toolkit available. Remember to specify the path to the `.toolkit` file. Registering a toolkit does not create an instance of the toolkit GUI.

```
toolkit_register <toolkit_file>
```

11.9.4 Launching a Toolkit

With the System Console, you can launch pre-registered toolkits in a number of ways:

- Click **Tools ➤ Toolkits**.
- Use the **Toolkits** tab. Each toolkit has a description, a detected hardware list, and a launch button.
- Use following command:

  ```
  toolkit_open <.toolkit_file_name>
  ```

You can launch a toolkit in the context of a hardware resource associated with a toolkit type. If you use the command:

```
toolkit_open <toolkit_name> <context>
```

the toolkit Tcl can retrieve the context by typing

```
set context [toolkit_get_context]
```

Related Links

toolkit_get_context on page 225

11.9.5 Matching Toolkits with IP Cores

You can match your toolkit with any IP core:

- When searching for IP, the toolkit looks for debug markers and matches IP cores to the toolkit requirements. In the toolkit file, use the requirement attribute to specify a debug type, as follows:

  ```
  <requirement><type>debug.type-name</type></requirement
  ```

- Create debug assignments in the `hw.tcl` for an IP core. `hw.tcl` files are available when you load the design in System Console.
- System Console discovers debug markers from identifiers in the hardware and associates with IP, without direct knowledge of the design.

11.9.6 Toolkit API

The Toolkit API service enables you to construct GUIs for visualizing and interacting with debug data. The Toolkit API is a graphical pane for the layout of your graphical widgets, which include buttons and text fields. Widgets pull data from other System Console services. Similarly, widgets use services to leverage user input to act on debug logic in your design.

Properties

Widget properties can push and pull information to the user interface. Widgets have properties specific to their type. For example, when you click a button, the button property `onClick` performs an action. A label widget does not have the same property, because the widget does not perform an action on click operation. However, both the button and label widgets have the `text` property to display text strings.

Layout

The Toolkit API service creates a widget hierarchy where the toolkit is at the top-level. The service implements group-type widgets that contain child widgets. Layout properties dictate layout actions that a parent performs on its children. For example, the `expandableX` property when set as `True`, expands the widget horizontally to encompass all of the available space. The `visible` property when set as `True` allows a widget to display in the GUI.

User Input

Some widgets allow user interaction. For example, the `textField` widget is a text box that allows user entries. Access the contents of the box with the `text` property. A Tcl script can either get or set the contents of the `textField` widget with the `text` property.

Callbacks

Some widgets perform user-specified actions, referred to as callbacks. The `textField` widget has the `onChange` property, which is called when text contents change. The `button` widget has the `onClick` property, which is called when you click a button. Callbacks update widgets or interact with services based on the contents of a text field, or the state of any other widget.

11.9.6.1 Customizing Toolkit API Widgets

Use the `toolkit_set_property` command to interact with the widgets that you instantiate. The `toolkit_set_property` command is most useful when you change part of the execution of a callback.

11.9.6.2 Toolkit API Script Examples

Example 26. Making the Toolkit Visible in System Console

Use the `toolkit_set_property` command to modify the `visible` property of the root toolkit. Use the word `self` if a property is applied to the entire toolkit. In other cases, refer to the root toolkit using `all`.

```
toolkit_set_property self visible true
```

Example 27. Adding Widgets

Use the `toolkit_add` command to add widgets.

```
toolkit_add my_button button all
```

The following commands add a label widget `my_label` to the root toolkit. In the GUI, the label appears as **Widget Label**.

```
set name "my_label"
set content "Widget Label"
toolkit_add $name label all
toolkit_set_property $name text $content
```

In the GUI, the displayed text changes to the new value. Add one more label:

```
toolkit_add my_label_2 label all
toolkit_set_property my_label_2 text "Another label"
```

The new label appears to the right of the first label.

To place the new label under the first, use the following command:

```
toolkit_set_property self itemsPerRow 1
```

Example 28. Gathering Input

To incorporate user input into your Toolkit API,

1. Create a text field using the following commands:

   ```
   set name "my_text_field"
   set widget_type "textField"
   set parent "all"
   toolkit_add $name $widget_type $parent
   ```

2. The widget size is very small. To make the widget fill the horizontal space, use the following command:

   ```
   toolkit_set_property my_text_field expandableX true
   ```

3. Now, the text field is fully visible. You can type text into the field, on clicking. To retrieve the contents of the field, use the following command:

   ```
   set content [toolkit_get_property my_text_field text]
   puts $content
   ```

This command prints the contents into the console.

Example 29. Updating Widgets Upon User Events

When you use callbacks, the Toolkit API can also perform actions without interactive typing:

1. Start by defining a procedure that updates the first label with the text field contents:

   ```
   proc update_my_label_with_my_text_field{
       set content [toolkit_get_property my_text_field text]
       toolkit_set_property my_label text $content
   }
   ```

2. Run the `update_my_label_with_my_text_field` command in the Tcl Console. The first label now matches the text field contents.

3. Use the `update_my_label_with_my_text_field` command whenever the text field changes:

   ```
   toolkit_set_property my_text_field onChange
   update_my_label_with_my_text_field
   ```

   The Toolkit executes the `onChange` property each time the text field changes. The execution of this property changes the first field to match what you type.

Example 30. Buttons

Use buttons to trigger actions.

1. To create a button that changes the second label:

   ```
   proc append_to_my_label_2 {suffix} {
                           set old_text [toolkit_get_property my_label_2 text]
                           set new_text "${old_text}${suffix}"
                           toolkit_set_property my_label_2 text $new_text
                           }
   ```

```
                        set text_to_append ", and more"
                        toolkit_add my_button button all
                        toolkit_set_property my_button onClick
[append_to_my_label_2 $text_to_append]
```

2. Click the button to append some text to the second label.

Example 31. Groups

The property `itemsPerRow` dictates the laying out of widgets in a group. For more complicated layouts where the number of widgets per row is different, use nested groups. To add a new group with more widgets per row:

```
toolkit_add my_inner_group group all
toolkit_set_property my_inner_group itemsPerRow 2
toolkit_add inner_button_1 button my_inner_group
toolkit_add inner_button_2 button my_inner_group
```

These commands create a row with a group of two buttons. To make the nested group more seamless, remove the border with the group name using the following commands:

```
toolkit_set_property my_inner_group title ""
```

You can set the `title` property to any other string to ensure the display of the border and title text.

Example 32. Tabs

Use tabs to manage widget visibility:

```
toolkit_add my_tabs tabbedGroup all
toolkit_set_property my_tabs expandableX true
toolkit_add my_tab_1 group my_tabs
toolkit_add my_tab_2 group my_tabs
toolkit_add tabbed_label_1 label my_tab_1
toolkit_add tabbed_label_2 label my_tab_2
toolkit_set_property tabbed_label_1 text "in the first tab"
toolkit_set_property tabbed_label_2 text "in the second tab"
```

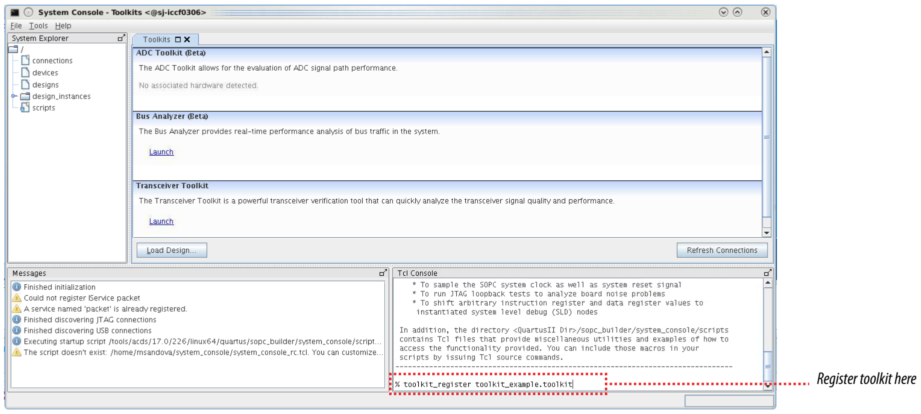
These commands add a set of two tabs, each with a group containing a label. Clicking on the tabs changes the displayed group/label.

11.9.6.3 Toolkit API GUI Example

Perform the following steps to register and launch a toolkit containing an interactive GUI window.

1. Write a toolkit description file. For a working example, refer to *Creating a Toolkit Description File*.
2. Generate a .tcl file using the text on *Toolkit API GUI Example .tcl File*.
3. Open the System Console.
4. Register your toolkit in the **Tcl Console** pane. Don't forget to include the relative path to your file's location.

Figure 123. Registering Your Toolkit



The ADC Toolkit allows for the evaluation of ADC signal path performance.

No associated hardware detected.

The Bus Analyzer provides real-time performance analysis of bus traffic in the system.

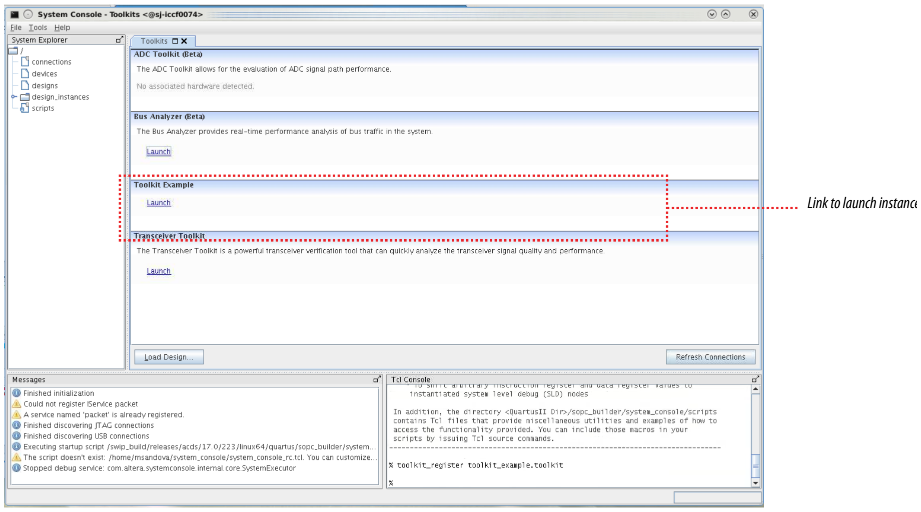
The Transceiver Toolkit is a powerful transceiver verification tool that can quickly analyze the transceiver signal quality and performance.

```
Finished initialization
Could not register IService packet
A service named 'packet' is already registered.
Finished discovering JTAG connections
Finished discovering USB connections
Executing startup script /tools/acds/17.0/226/linux64/quartus/sopc_builder/system_console/script...
The script doesn't exist: /home/msandova/system_console/system_console_rc.tcl. You can customize...
```

```
   * To sample the SOPC system clock as well as system reset signal
   * To run JTAG loopback tests to analyze board noise problems
   * To shift arbitrary instruction register and data register values to
    instantiated system level debug (SLD) nodes

In addition, the directory <QuartusII Dir>/sopc_builder/system_console/scripts
contains Tcl files that provide miscellaneous utilities and examples of how to
access the functionality provided. You can include those macros in your
scripts by issuing Tcl source commands.
------------------------------------------------------------------------------
% toolkit_register toolkit_example.toolkit
```

The Toolkit appears in the **Toolkits** tab

Figure 124. Toolkits Tab After Toolkit Example Registration



The ADC Toolkit allows for the evaluation of ADC signal path performance.

No associated hardware detected.

The Bus Analyzer provides real-time performance analysis of bus traffic in the system.

The Transceiver Toolkit is a powerful transceiver verification tool that can quickly analyze the transceiver signal quality and performance.

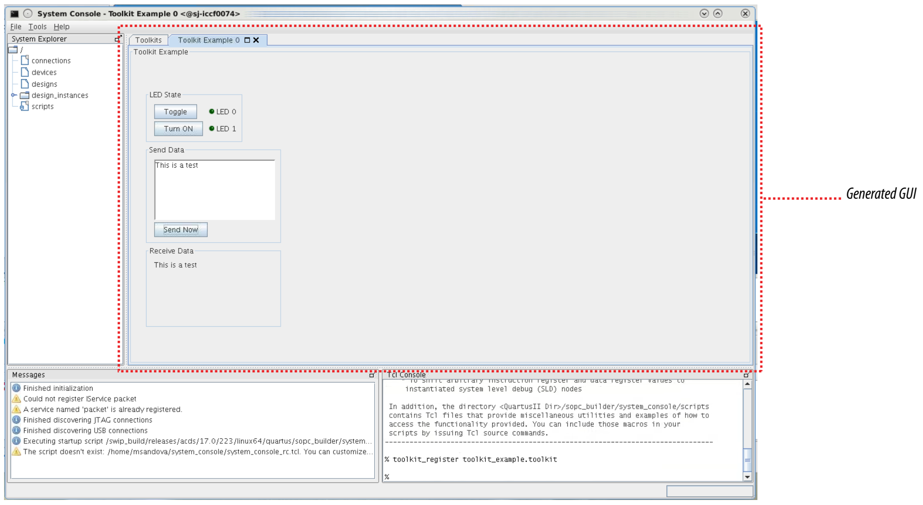
```
Finished initialization
Could not register IService packet
A service named 'packet' is already registered.
Finished discovering JTAG connections
Finished discovering USB connections
Executing startup script /swip_build/releases/acds/17.0/223/linux64/quartus/sopc_builder/system...
The script doesn't exist: /home/msandova/system_console/system_console_rc.tcl. You can customize...
Stopped debug service: com.altera.systemconsole.internal.core.SystemExecutor
```

```
    instantiated system level debug (SLD) nodes

In addition, the directory <QuartusII Dir>/sopc_builder/system_console/scripts
contains Tcl files that provide miscellaneous utilities and examples of how to
access the functionality provided. You can include those macros in your
scripts by issuing Tcl source commands.
------------------------------------------------------------------------------

% toolkit_register toolkit_example.toolkit

%
```

5. Click the Launch link.

   A new tab appears, containing the widgets you specified in the TCL file.

Figure 125. Toolkit Example GUI



```
Finished initialization
Could not register IService packet
A service named 'packet' is already registered.
Finished discovering JTAG connections
Finished discovering USB connections
Executing startup script /swip_build/releases/acds/17.0/223/linux64/quartus/sopc_builder/system...
The script doesn't exist: /home/msandova/system_console/system_console_rc.tcl. You can customize...
```

```
    instantiated system level debug (SLD) nodes

 In addition, the directory <QuartusII Dir>/sopc_builder/system_console/scripts
 contains Tcl files that provide miscellaneous utilities and examples of how to
 access the functionality provided. You can include those macros in your
 scripts by issuing Tcl source commands.
------------------------------------------------------------------------------

% toolkit_register toolkit_example.toolkit

%
```

When you insert text in the **Send Data** field and click **Launch**, the text appears in the **Receive Data** field.

Related Links

Creating a Toolkit Description File on page 212

11.9.6.3.1 Toolkit API GUI Example .tcl File

The following Toolkit API .tcl file creates a GUI window that provides debug interaction with your design.

```
namespace eval Test {

    variable ledValue 0
    variable dashboardActive 0
    variable Switch_off 1

    proc toggle { position } {
        set ::Test::ledValue ${position}
        ::Test::updateDashboard
        }

    proc sendText {} {
        set sendText [toolkit_get_property sendTextText text]
        toolkit_set_property receiveTextText text $sendText
        }

    proc dashBoard {} {

        if { ${::Test::dashboardActive} == 1 } {
            return -code ok "dashboard already active"
        }

        set ::Test::dashboardActive 1
        #
        # top group widget
        #
        toolkit_add  topGroup group self
        toolkit_set_property  topGroup expandableX false
        toolkit_set_property  topGroup expandableY false
        toolkit_set_property  topGroup itemsPerRow 1
        toolkit_set_property  topGroup title ""
```

```
#
# leds group widget
#
toolkit_add  ledsGroup group topGroup
toolkit_set_property  ledsGroup expandableX false
toolkit_set_property  ledsGroup expandableY false
toolkit_set_property  ledsGroup itemsPerRow 2
toolkit_set_property  ledsGroup title "LED State"

#
# leds widgets
#
toolkit_add  led0Button button ledsGroup
toolkit_set_property  led0Button enabled true
toolkit_set_property  led0Button expandableX false
toolkit_set_property  led0Button expandableY false
toolkit_set_property  led0Button text "Toggle"
toolkit_set_property  led0Button onClick {::Test::toggle 1}

toolkit_add  led0LED led ledsGroup
toolkit_set_property  led0LED expandableX false
toolkit_set_property  led0LED expandableY false
toolkit_set_property  led0LED text "LED 0"
toolkit_set_property  led0LED color "green_off"

toolkit_add  led1Button button ledsGroup
toolkit_set_property  led1Button enabled true
toolkit_set_property  led1Button expandableX false
toolkit_set_property  led1Button expandableY false
toolkit_set_property  led1Button text "Turn ON"
toolkit_set_property  led1Button onClick {::Test::toggle 2}

toolkit_add  led1LED led ledsGroup
toolkit_set_property  led1LED expandableX false
toolkit_set_property  led1LED expandableY false
toolkit_set_property  led1LED text "LED 1"
toolkit_set_property  led1LED color "green_off"


#
# sendText widgets
#
toolkit_add  sendTextGroup group topGroup
toolkit_set_property  sendTextGroup expandableX false
toolkit_set_property  sendTextGroup expandableY false
toolkit_set_property  sendTextGroup itemsPerRow 1
toolkit_set_property  sendTextGroup title "Send Data"

toolkit_add  sendTextText text sendTextGroup
toolkit_set_property  sendTextText expandableX false
toolkit_set_property  sendTextText expandableY false
toolkit_set_property  sendTextText preferredWidth 200
toolkit_set_property  sendTextText preferredHeight 100
toolkit_set_property  sendTextText editable true
toolkit_set_property  sendTextText htmlCapable false
toolkit_set_property  sendTextText text ""

toolkit_add  sendTextButton button sendTextGroup
toolkit_set_property  sendTextButton enabled true
toolkit_set_property  sendTextButton expandableX false
toolkit_set_property  sendTextButton expandableY false
toolkit_set_property  sendTextButton text "Send Now"
toolkit_set_property  sendTextButton onClick {::Test::sendText}

#
# receiveText widgets
#
toolkit_add  receiveTextGroup group topGroup
toolkit_set_property  receiveTextGroup expandableX false
toolkit_set_property  receiveTextGroup expandableY false
toolkit_set_property  receiveTextGroup itemsPerRow 1
```

```
        toolkit_set_property  receiveTextGroup title "Receive Data"

        toolkit_add  receiveTextText text receiveTextGroup
        toolkit_set_property  receiveTextText expandableX false
        toolkit_set_property  receiveTextText expandableY false
        toolkit_set_property  receiveTextText preferredWidth 200
        toolkit_set_property  receiveTextText preferredHeight 100
        toolkit_set_property  receiveTextText editable false
        toolkit_set_property  receiveTextText htmlCapable false
        toolkit_set_property  receiveTextText text ""

        return -code ok
    }

    proc updateDashboard {} {

        if { ${::Test::dashboardActive} > 0 } {

                toolkit_set_property  ledsGroup title "LED State"
                if { [ expr ${::Test::ledValue} & 0x01 &  \
                            ${::Test::Switch_off} ] } {
                    toolkit_set_property  led0LED color "green"
                    set ::Test::Switch_off  0
                } else {
                    toolkit_set_property  led0LED color "green_off"
                    set ::Test::Switch_off  1
                }
                if { [ expr ${::Test::ledValue} & 0x02 ] } {
                    toolkit_set_property  led1LED color "green"
                } else {
                    toolkit_set_property  led1LED color "green_off"
                }
        }
    }
}
::Test::dashBoard
```

11.9.6.4 Toolkit API Commands

Toolkit API commands run in the context of a unique toolkit instance.

toolkit_register on page 222

toolkit_open on page 223

get_quartus_ini on page 224

toolkit_get_context on page 225

toolkit_get_types on page 226

toolkit_get_properties on page 227

toolkit_add on page 228

toolkit_get_property on page 229

toolkit_set_property on page 230

toolkit_remove on page 231

toolkit_get_widget_dimensions on page 232

11.9.6.4.1 toolkit_register

Description

Point to the XML file that describes the plugin (`.toolkit` file) .

Usage

`toolkit_register` *<toolkit_file>*

Returns

No return value.

Arguments

<toolkit_file>	Path to the toolkit definition file.

Example

```
toolkit_register /path/to/toolkit_example.toolkit
```

11.9.6.4.2 toolkit_open

Description

Opens an instance of a toolkit in System Console.

Usage

`toolkit_open` *<toolkit_id> [<context>]*

Returns

No return value.

Arguments

<toolkit_id>	Name of the toolkit type to open.
<context>	An optional context, such as a service path for a hardware resource that is associated with the toolkit that opens.

Example

```
toolkit_open my_toolkit_id
```

11.9.6.4.3 get_quartus_ini

Description

Returns the value of an `ini` setting from the Intel Quartus Prime software `.ini` file.

Usage

`get_quartus_ini` *<ini> <type>*

Returns

Value of `ini` setting.

Arguments

<ini>	Name of the Intel Quartus Prime software `.ini` setting.
<type>	(Optional) Type of `.ini` setting. The known types are `string` and `enabled`. If the type is `enabled`, the value of the `.ini` setting returns `1`, or `0` if not enabled.

Example

```
set my_ini_enabled [get_quartus_ini my_ini enabled]
```

```
set my_ini_raw_value [get_quartus_ini my_ini]
```

11.9.6.4.4 toolkit_get_context

Description

Returns the context that was specified when the toolkit was opened. If no context was specified, returns an empty string.

Usage

`toolkit_get_context`

Returns

Context.

Arguments

No arguments.

Example

```
set context [toolkit_get_context]
```

11.9.6.4.5 toolkit_get_types

Description

Returns a list of widget types.

Usage

`toolkit_get_types`

Returns

List of widget types.

Arguments

No arguments.

Example

```
set widget_names [toolkit_get_types]
```

11.9.6.4.6 toolkit_get_properties

Description

Returns a list of toolkit properties for a type of widget.

Usage

`toolkit_get_properties` <*widgetType*>

Returns

List of toolkit properties.

Arguments

<*widgetType*>	Type of widget.

Example

```
set widget_properties [toolkit_get_properties xyChart]
```

11.9.6.4.7 toolkit_add

Description

Adds a widget to the current toolkit.

Usage

`toolkit_add` *<id>* *<type>*<*groupid*>

Returns

No return value.

Arguments

<id>	A unique ID for the widget being added.
<type>	The type of widget that is being added.
<groupid>	The ID for the parent group that will contain the new widget. Use `self` for the toolkit base group.

Example

```
toolkit_add my_button button parentGroup
```

11.9.6.4.8 toolkit_get_property

Description

Returns the property value for a specific widget.

Usage

`toolkit_get_property` *<id>* *<propertyName>*

Returns

The property value.

Arguments

<id>	A unique ID for the widget being queried.
<propertyName>	The name of the widget property.

Example

```
set enabled [toolkit_get_property my_button enabled]
```

11.9.6.4.9 toolkit_set_property

Description

Sets the property value for a specific widget.

Usage

`toolkit_set_property` <id><*propertyName*> <*value*>

Returns

No return value.

Arguments

<*id*>	A unique ID for the widget being modified.
<*propertyName*>	The name of the widget property being set.
<*value*>	The new value for the widget property.

Example

```
toolkit_set_property my_button enabled 0
```

11.9.6.4.10 toolkit_remove

Description

Removes a widget from the specified toolkit.

Usage

`toolkit_remove` <*id*>

Returns

No return value.

Arguments

<*id*>	A unique ID for the widget being removed.

Example

```
toolkit_remove my_button
```

11.9.6.4.11 toolkit_get_widget_dimensions

Description

Returns the width and height of the specified widget.

Usage

`toolkit_get_widget_dimensions` <*id*>

Returns

Width and height of specified widget.

Arguments

<*id*>	A unique ID for the widget being added.

Example

```
set dimensions [toolkit_get_widget_dimensions my_button]
```

11.9.6.5 Toolkit API Properties

The following are the Toolkit API widget properties:

11.9.6.5.1 Widget Types and Properties

Table 58. Toolkit API Widget Types and Properties

Name	Description
`enabled`	Enables or disables the widget.
`expandable`	Controls whether the widget is expandable.
`expandableX`	Allows the widget to resize horizontally if there is space available in the cell where it resides.
`expandableY`	Allows the widget to resize vertically if there is space available in the cell where it resides.
`foregroundColor`	Sets the foreground color.
`maxHeight`	If the widget's `expandableY` is set, this is the maximum height in pixels that the widget can take.
`minHeight`	If the widget's `expandableY` is set, this is the minimum height in pixels that the widget can take.
`maxWidth`	If the widget's `expandableX` is set, this is the maximum width in pixels that the widget can take.
`minWidth`	If the widget's `expandableX` is set, this is the minimum width in pixels that the widget can take.
`preferredHeight`	The height of the widget if `expandableY` is not set.
`preferredWidth`	The width of the widget if `expandableX` is not set.
`toolTip`	Implements a mouse-over tooltip.
`visible`	Displays the widget.

11.9.6.5.2 barChart Properties

Table 59. Toolkit API barChart Properties

Name	Description
`title`	Chart title.
`labelX`	X-axis label text.
`label`	X-axis label text.
`range`	Y-axis value range. By default, it is auto range. Specifiy the range using a Tcl list, for example: `[list lower_numerical_value upper_numerical_value]`.
`itemValue`	Specify the value using a Tcl list, for example: `[list bar_category_str numerical_value]`.

11.9.6.5.3 button Properties

Table 60. Toolkit API button Properties

Name	Description
`onClick`	Specifies the Tcl command to run every time you click the button. Usually the command is a `proc`.
`text`	The text on the button.

11.9.6.5.4 checkBox Properties

Table 61. Toolkit API checkBox Properties

Name	Description
`checked`	Specifies the state of the checkbox.
`onClick`	Specifies the Tcl command to run every time you click the checkbox. The command is usually a `proc`.
`text`	The text on the checkbox.

11.9.6.5.5 comboBox Properties

Table 62. Toolkit API comboBox Properties

Name	Description
`onChange`	A Tcl callback to run when the value of the combo box changes.
`options`	A list of items to display in the combo box.
`selectedItem`	The selected item in the combo box.

11.9.6.5.6 dial Properties

Table 63. Toolkit API dial Properties

Name	Description
`max`	The maximum value that the dial can show.
`min`	The minimum value that the dial can show.
`ticksize`	The space between the different tick marks of the dial.
`title`	The title of the dial.
`value`	The value that the dial's needle marks. It must be between min and max.

11.9.6.5.7 fileChooserButton Properties

Table 64. Toolkit API fileChooserButton Properties

Name	Description
`text`	The text on the button.
`onChoose`	A Tcl command that runs every time you click the button. The command is usually a `proc`.
`title`	The title of the dialog box.
`chooserButtonText`	The text of the dialog box approval button. Default value is `Open`.
`filter`	The file filter, based on extension. The filter supports only one extension. By default, the filter allows all file names. Specify the filter using the syntax `[list filter_description file_extension]`, for example: `[list "Text Document (.txt)" "txt"]`.
`mode`	Specifies what kind of files or directories you can select. The default is `files_only`. Possible options are `files_only` and `directories_only`.
`multiSelectionEnabled`	Controls whether you can select multiple files. Default value is `false`.
`paths`	This property is read-only. Returns a list of file paths selected in the file chooser dialog box. The property is most useful when you use it within the `onClick` script, or inside a procedure that updates the result after the dialog box closes.

11.9.6.5.8 group Properties

Table 65. Toolkit API group Properties

Name	Description
`itemsPerRow`	The number of widgets the group can position in one row, from left to right, before moving to the next row.
`title`	The title of the group. Groups with a title can have a border around them, and setting an empty title removes the border.

11.9.6.5.9 label Properties

Table 66. Toolkit API label Properties

Name	Description
`text`	The text to show in the label.

11.9.6.5.10 led Properties

Table 67. Toolkit API led Properties

Name	Description
`color`	The color of the LED. The options are: `red_off`, `red`, `yellow_off`, `yellow`, `green_off`, `green`, `blue_off`, `blue`, and `black`.
`text`	The text to show next to the LED.

11.9.6.5.11 lineChart Properties

Table 68. Toolkit API lineChart Properties

Name	Description
`title`	Chart title.
`labelX`	X-axis label text.
`labelY`	Y-axis label text.
`range`	Y-axis value range. By default, it is auto range. Specify the range using a Tcl list, for example: `[list lower_numerical_value upper_numerical_value]`.
`itemValue`	Item value. Specifiy the value using a Tcl list, for example: `[list bar_category_str numerical_value]`.

11.9.6.5.12 list Properties

Table 69. Toolkit API list Properties

Name	Description
`selected`	Index of the selected item in the combo box.
`options`	List of options to display.
`onChange`	A Tcl callback to run when the selected item in the list changes.

11.9.6.5.13 pieChart Properties

Table 70. Toolkit API pieChart Properties

Name	Description
`title`	Chart title.
`itemValue`	Item value. Specified using a Tcl list, for example: `[list bar_category_str numerical_value]`.

11.9.6.5.14 table Properties

Table 71. Toolkit API table Properties

Name	Description
`columnCount`	The number of columns (Mandatory) (`0`, by default).
`rowCount`	The number of rows (Mandatory) (`0`, by default).
`headerReorderingAllowed`	Controls whether you can drag the columns (`false`, by default).
`headerResizingAllowed`	Controls whether you can resize all column widths. (`false`, by default). *Note:* You can resize each column individually with the `columnWidthResizable` property.
`rowSorterEnabled`	Controls whether you can sort the cell values in a column (`false`, by default).
`showGrid`	Controls whether to draw both horizontal and vertical lines (`true`, by default).
`showHorizontalLines`	Controls whether to draw horizontal line (`true`, by default).
`rowIndex`	Current row index. Zero-based. This value affects some properties below (`0`, by default).
`columnIndex`	Current column index. Zero-based. This value affects all column specific properties below (`0`, by default).
`cellText`	Specifies the text inside the cell given by the current `rowIndex` and `columnIndex` (Empty, by default).
`selectedRows`	Control or retrieve row selection.
`columnHeader`	The text in the column header.
`columnHeaders`	A list of names to define the columns for the table.
`columnHorizontalAlignment`	The cell text alignment in the specified column. Supported types are `leading` (default), `left`, `center`, `right`, `trailing`.
`columnRowSorterType`	The type of sorting method. This is applicable only if `rowSorterEnabled` is `true`. Each column has its own sorting type. Possible types are `string` (default), `int`, and `float`.
`columnWidth`	The number of pixels in the column width.
`columnWidthResizable`	Controls whether the column width is resizable by you (`false`, by default).
`contents`	The contents of the table as a list. For a table with columns A, B, and C, the format of the list is `{A1 B1 C1 A2 B2 C2 ...}`.

11.9.6.5.15 text Properties

Table 72. Toolkit API text Properties

Name	Description
`editable`	Controls whether the text box is editable.
`htmlCapable`	Controls whether the text box can format HTML.
`text`	The text to show in the text box.

11.9.6.5.16 textField Properties

Table 73. Toolkit API textField Properties

Name	Description
`editable`	Controls whether the text box is editable.
`onChange`	A Tcl callback to run when you change the content of the text box.
`text`	The text in the text box.

11.9.6.5.17 timeChart Properties

Table 74. Toolkit API timeChart Properties

Name	Description
`labelX`	The label for the X-axis.
`labelY`	The label for the Y-axis.
`latest`	The latest value in the series.
`maximumItemCount`	The number of sample points to display in the historic record.
`title`	The title of the chart.
`range`	Sets the range for the chart. The range has the form `{low, high}`. The low/high values are doubles.
`showLegend`	Specifies whether a legend for the series is shown in the graph.

11.9.6.5.18 xyChart Properties

Table 75. Toolkit API xyChart Properties

Name	Properties
`title`	Chart title.
`labelX`	X-Axis label text.
`labelY`	Y-Axis label text.
`range`	Sets the range for the chart. The range is of the form `{low, high}`. The low/high values are doubles.
`maximumItemCount`	Specifies the maximum number of data values to keep in a data series. This setting only affects new data in the chart. If you add more data values than the `maximumItemCount`, only the last `maximumItemCount` number of entries are kept.
`series`	Adds a series of data to the chart. The first value in the spec is the identifier for the series. If the same identifier is set twice, the Toolkit API selects the most recent series. If the identifier does not contain series data, that series is removed from the chart. Specify the series in a Tcl list: `{identifier, x-1 y-1, x-2 y-2}`.
`showLegend`	Sets whether a legend for the series appears in the graph.

11.10 ADC Toolkit

The ADC Toolkit is designed to work with Intel MAX 10 devices and helps you understand the performance of the analog signal chain as seen by the on-board ADC hardware. The GUI displays the performance of the ADC using industry standard metrics. You can export the collected data to a `.csv` file and process this raw data yourself. The ADC Toolkit is built on the System Console framework and can only be operated using the GUI. There is no Tcl support for the tool.

Prerequisites for Using the ADC Toolkit

- Altera Modular ADC IP core
  - **External Reference Voltage** if you select **External** in the Altera Modular ADC IP parameters
- Reference signal

The ADC Toolkit needs a sine wave signal to be fed to the analog inputs. You need the capability to precisely set the level and frequency of the reference signal. A high-precision sine wave is needed for accurate test results; however, there are useful things that can be read in **Scope** mode with any input signal.

To achieve the best testing results, the reference signal should have less distortions than the device ADC is able to resolve. If this is not the case, then you will be adding distortions from the source into the resulting ADC distortion measurements. The limiting factor is based on hardware precision.

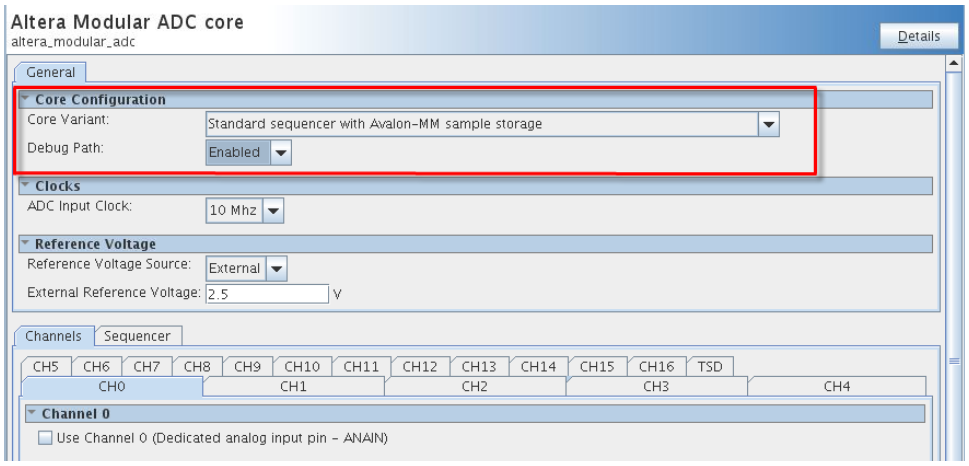
Note: When applying a sine wave, the ADC should sample at 2x the fundamental sine wave frequency. There should be a low-pass filter, 3dB point set to the fundamental frequency.

Configuring the Altera Modular ADC IP Core

The Altera Modular ADC IP core needs to be included in your design. You can instantiate this IP core from the **IP Catalog**. When you configure this IP core in the **Parameter Editor**, you need to enable the **Debug Path** option located under **Core Configuration**.

There are two limitations in the Intel Quartus Prime software v14.1 for the Altera Modular ADC IP core. The ADC Toolkit does not support the **ADC control core only** option under **Core Configuration**. You must select a core variant that uses the standard sequencer in order for the Altera Modular ADC IP core to work with ADC Toolkit. Also, if an Avalon Master is not connected to the sequencer, you must manually start the sequencer before the ADC Toolkit will work.

Figure 126. Altera Modular ADC Core


Altera Modular ADC core
altera_modular_adc

Core Configuration
Core Variant: Standard sequencer with Avalon-MM sample storage
Debug Path: Enabled

Clocks
ADC Input Clock: 10 Mhz

Reference Voltage
Reference Voltage Source: External
External Reference Voltage: 2.5 V

Channel 0
Use Channel 0 (Dedicated analog input pin - ANAIN)

Starting the ADC Toolkit

You can launch the ADC Toolkit from System Console. Before starting the ADC toolkit, you need to verify that your board is programmed. You can then load your `.sof` by clicking **File ➤ Load Design**. If System Console was started with an active project, your design is auto-loaded when you start System Console.

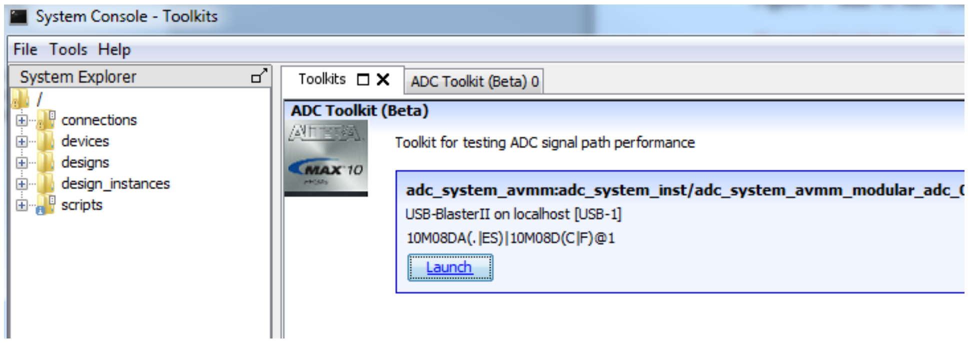
There are two methods to start the ADC Toolkit. Both methods require you to have a Intel MAX 10 device connected, programmed with a project, and linked to this project. However, the **Launch** command only shows up if these requirements are met. You can always start the ADC Toolkit from the **Tools** menu, but if the above requirements are not met, no connection will be made.

- Click **Tools ➤ ADC Toolkit**
- Alternatively, click **Launch** from the **Toolkits** tab. The path for the device is displayed above the **Launch** button.

Note: Only one ADC Toolkit enabled device can be connected at a time.

Upon starting the ADC Toolkit, an identifier path on the ADC Toolkit tab shows you which ADC on the device is being used for this instance of the ADC Toolkit.

Figure 127. Launching ADC Toolkit



Toolkit for testing ADC signal path performance

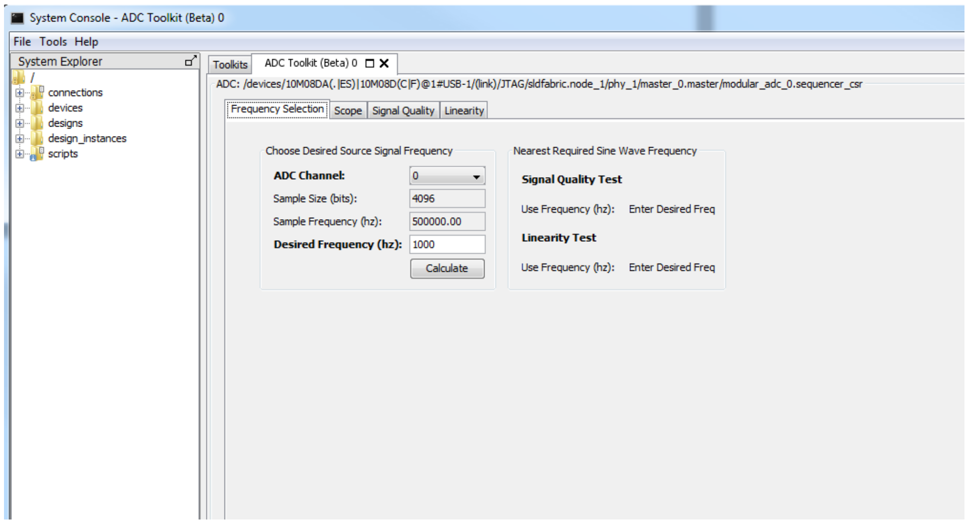
adc_system_avmm:adc_system_inst/adc_system_avmm_modular_adc_
USB-BlasterII on localhost [USB-1]
10M08DA(.|ES)|10M08D(C|F)@1

ADC Toolkit Flow

The ADC Toolkit GUI consists of four panels: **Frequency Selection**, **Scope**, **Signal Quality**, and **Linearity**.

1. Use the **Frequency Selection** panel to calculate the required sine wave frequency for proper signal quality testing. The ADC Toolkit will give you the nearest ideal frequency based on your desired reference signal frequency.
2. Use the **Scope** panel to tune your signal generator or inspect input signal characteristics.
3. Use the **Signal Quality** panel to test the performance of your ADC using industry standard metrics.
4. Use the **Linearity** panel to test the linearity performance of your ADC and display differential and integral non-linearity results.

Figure 128. ADC Toolkit GUI



ADC: /devices/10M08DA(.|ES)|10M08D(C|F)@1#USB-1/(link)/JTAG/sldfabric.node_1/phy_1/master_0.master/modular_adc_0.sequencer_csr

Choose Desired Source Signal Frequency

ADC Channel:	0
Sample Size (bits):	4096
Sample Frequency (hz):	500000.00
Desired Frequency (hz):	1000

Nearest Required Sine Wave Frequency

Signal Quality Test
Use Frequency (hz): Enter Desired Freq
Linearity Test
Use Frequency (hz): Enter Desired Freq

Related Links

- Using the ADC Toolkit in Intel MAX 10 Devices online training
- Intel MAX 10 FPGA Device Overview
- Intel MAX 10 FPGA Device Datasheet

- Intel MAX 10 FPGA Design Guidelines
- Intel MAX 10 Analog to Digital Converter User Guide
- Additional information about sampling frequency
  Nyquist sampling theorem and how it relates to the nominal sampling interval required to avoid aliasing.

11.10.1 ADC Toolkit Terms

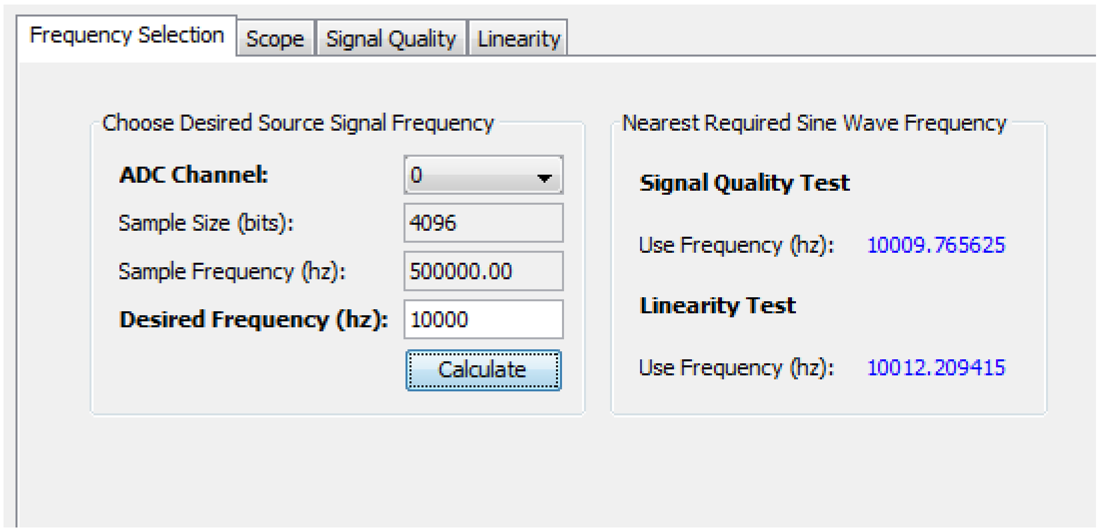
Table 76. ADC Toolkit Terms

Term	Description
SNR	The ratio of the output signal voltage level to the output noise level.
THD	The ratio of the sum of powers of the harmonic frequency components to the power of the fundamental/original frequency component.
SFDR	Characterizes the ratio between the fundamental signal and the highest spurious in the spectrum.
SINAD	The ratio of the RMS value of the signal amplitude to the RMS value of all other spectral components, including harmonics, but excluding DC.
ENOB	The number of bits with which the ADC behaves.
DNL	The maximum and minimum difference in the step width between actual transfer function and the perfect transfer function
INL	The maximum vertical difference between the actual and the ideal curve. It indicates the amount of deviation of the actual curve from the ideal transfer curve.

11.10.2 Setting the Frequency of the Reference Signal

You use the **Frequency Selection** panel to compute the required reference signal frequency to run the ADC performance tests. The sine wave frequency is critical and affects the validity of your test results.

Figure 129. Frequency Selection Panel



Choose Desired Source Signal Frequency

ADC Channel:	0
Sample Size (bits):	4096
Sample Frequency (hz):	500000.00
Desired Frequency (hz):	10000

Nearest Required Sine Wave Frequency

Signal Quality Test

Use Frequency (hz): 10009.765625

Linearity Test

Use Frequency (hz): 10012.209415

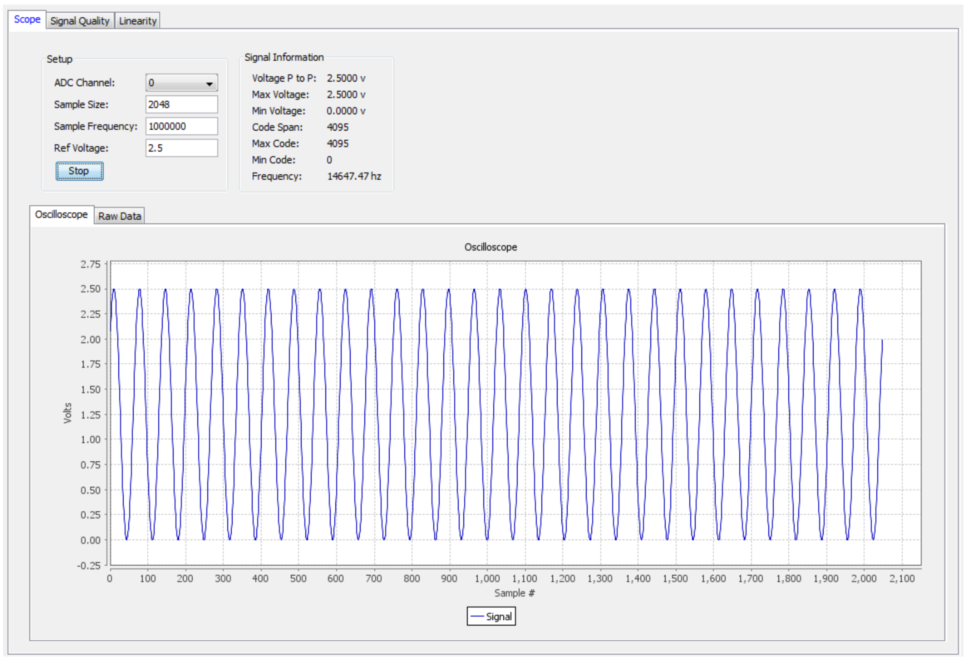
To set the frequency of the reference signal:

1. On **ADC Channel**, select the ADC channel that you plan to test.
   The tool populates the **Sample Size** and **Sample Frequency** fields.
2. Enter the **Desired Frequency**. This is your desired frequency for testing. You need to complete this procedure to calculate the frequency that you set your signal generator to, which will differ depending on the type of test you want to do with the ADC Toolkit.
3. Click **Calculate**.
   - The closest frequency for valid testing near your desired frequency displays under both **Signal Quality Test** and **Linearity Test**.
   - The nearest required sine wave frequencies are different for the signal quality test and linearity test.
4. Set your signal generator to the precise frequency given by the tool based on the type of test you want to run.

11.10.3 Tuning the Signal Generator

You use the **Scope** panel to tune your signal generator in order to achieve the best possible performance from the ADC.

Figure 130. Scope Mode Panel



Setup

ADC Channel:	0
Sample Size:	2048
Sample Frequency:	1000000
Ref Voltage:	2.5

Signal Information

Voltage P to P:	2.5000 v
Max Voltage:	2.5000 v
Min Voltage:	0.0000 v
Code Span:	4095
Max Code:	4095
Min Code:	0
Frequency:	14647.47 hz

To tune your signal generator:

1. On **ADC Channel**, select the ADC channel that you plan to test.
2. Enter your reference **Sample Frequency** (unless the tool can extract this value from your IP).
3. Enter your **Ref Voltage** (unless the tool can extract this value from your IP).
4. Click **Run**.
   The tool will repeatedly capture a buffer worth of data and display the data as a waveform and display additional information under **Signal Information**.
5. Tune your signal generator to use the maximum dynamic range of the ADC without clipping. Avoid hitting 0 or 4095 because your signal will likely be clipping. Look at the displayed sine wave under **Oscilloscope** to see that the top and bottom peaks are evenly balanced to ensure you have selected the optimum value.
   - For Intel MAX 10 devices, you want to get as close to **Min Code = 0** and **Max Code = 4095** without actually hitting those values.
   - The frequency should be set precisely to the value needed for testing such that coherent sampling is observed in the test window. Before moving forward, follow the suggested value for signal quality testing or linearity testing, which is displayed next to the actual frequency that is detected.
   - From the **Raw Data** tab, you can export your data as a `.csv` file.

Related Links

Additional information about coherent sampling vs window sampling

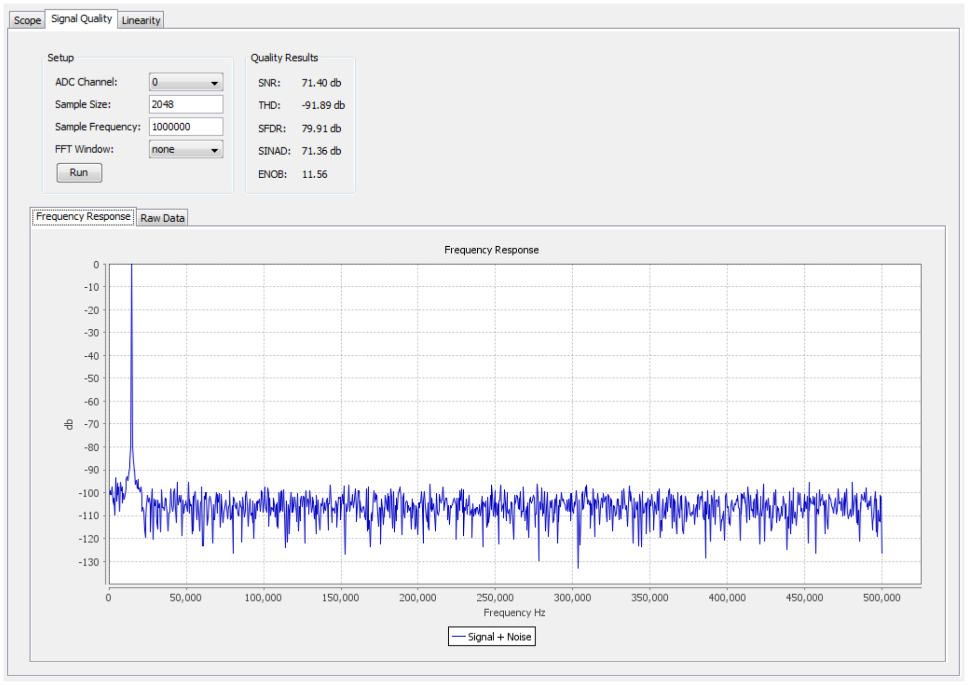
11.10.4 Running a Signal Quality Test

The available performance metrics in signal quality test mode are the following: signal to noise ratio (SNR), total harmonic distortion (THD), spurious free dynamic range (SFDR), signal to noise and distortion ratio (SINAD), effective number of bits (ENOB), and a frequency response graph.The frequency response graph shows the signal, noise floor, and any spurs or harmonics.

The signal quality parameters are measurements relative to the carrier signal and not the full scale of the ADC.

Before running a signal quality test, ensure that you have set up the frequency of the reference signal using **Scope** mode.

Figure 131. Signal Quality Panel



Setup

ADC Channel:	0
Sample Size:	2048
Sample Frequency:	1000000
FFT Window:	none

Quality Results

SNR:	71.40 db
THD:	-91.89 db
SFDR:	79.91 db
SINAD:	71.36 db
ENOB:	11.56

To run a signal quality test:

1. On **ADC Channel**, select the ADC channel that you plan to test.
2. Click **Run**.

   From the **Raw Data** tab, you can export your data as a `.csv` file.

For signal quality tests, the signal must be coherently sampled. Based on the sampling rate and number of samples to test, specific input frequencies are required for coherent sampling.The sample frequency for each channel is calculated based on the ADC sequencer configuration.

Related Links

Additional information about dynamic parameters such as SNR, THD, etc

11.10.5 Running a Linearity Test

The linearity test determines the linearity of the step sizes of each ADC code. It uses a histogram testing method which requires sinusoidal inputs which are easier to source from signal generators and DACs than other test methods.

When using **Linearity** test mode, your reference signal must meet specific requirements.

- The signal source covers the full code range of the ADC. Results improve if the time spent at code end is equivalent, by tuning the reference signal in **Scope** mode.
- You have to make sure if using code ends that you are not clipping the signal. Look at the signal in **Scope** mode to see that it does not look flat at the top or bottom. It may be desirable to back away from code ends and test a smaller range within the desired operating range of the ADC input signal.
- Choosing a frequency that is not an integer multiple of the sample rate and buffer size helps to ensure all code bins are filled relatively evenly to the probability density function of a sine wave. If an integer multiple is selected, some bins may be skipped entirely while others are over populated. This makes the tests results invalid. Use the frequency calculator feature to determine a good signal frequency near your desired frequency.

To run a linearity test:

1. On **ADC Channel**, select the ADC channel that you plan to test.
2. Enter the test sample size in **Burst Size**. Larger samples increase the confidence in the test results.
3. Click **Run**.
    - You can stop the test at anytime, as well as click **Run** again to continue adding to the aggregate data. To start fresh, click **Reset** after you stop a test. Anytime you change the input signal or channel, you should click **Reset** so your results are correct for a particular input.
    - There are three graphical views of the data: **Histogram** view, **DNL** view, and **INL** view.
    - From the **Raw Data** tab, you can export your data as a `.csv` file.

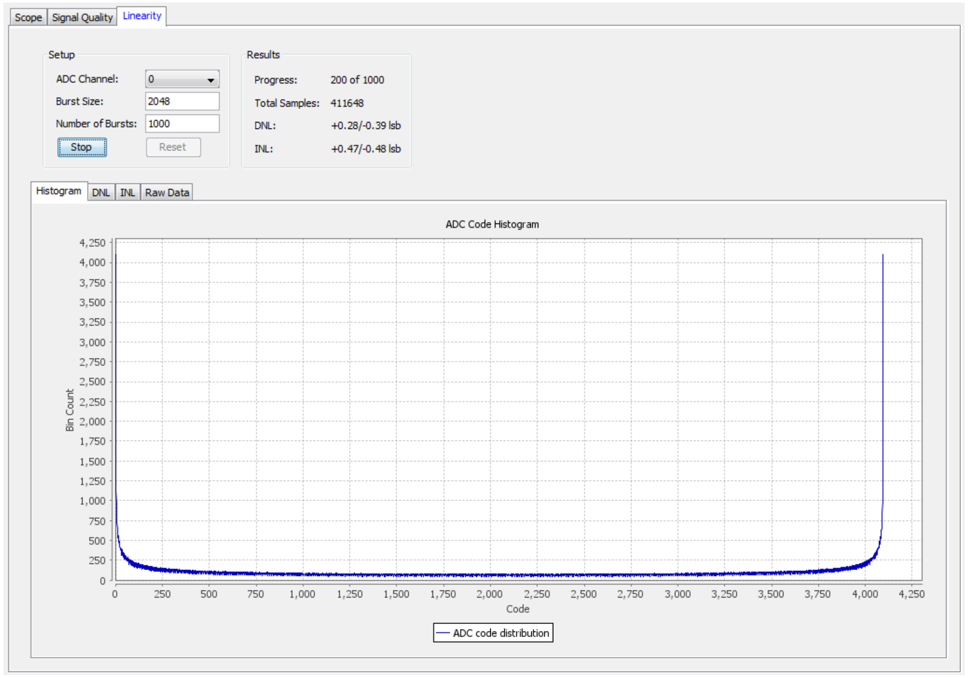
11.10.6 ADC Toolkit Data Views

Histogram View

The **Histogram** view shows how often each code appears. The graph updates every few seconds as it collects data. You can use the **Histogram** view to quickly check if your test signal is set up appropriately.

Figure 132. Example of Pure Sine Wave Histogram

The figure below shows the shape of a pure sine wave signal. Your reference signal should look similar.

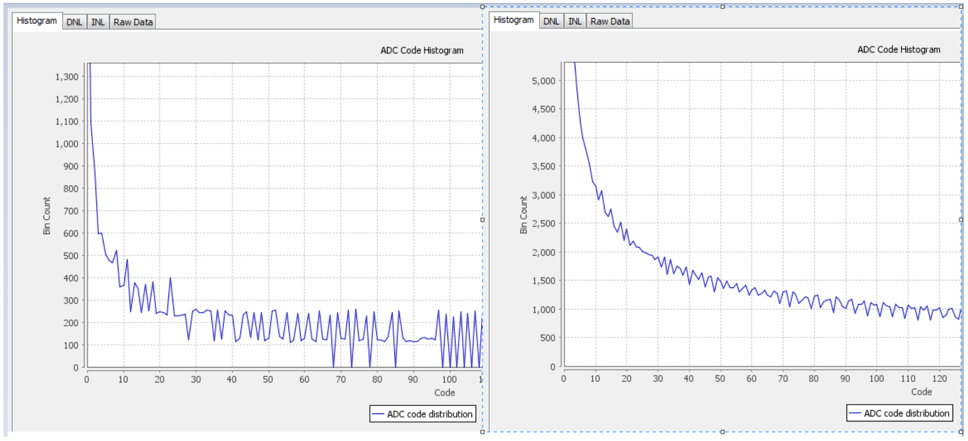


Setup
ADC Channel: 0
Burst Size: 2048
Number of Bursts: 1000

Results
Progress: 200 of 1000
Total Samples: 411648
DNL: +0.28/-0.39 lsb
INL: +0.47/-0.48 lsb

If your reference signal is not a relatively smooth line, but has jagged edges with some bins having a value of 0, and adjacent bins with a much higher value, then the test signal frequency is not adequate. Use **Scope** mode to help choose a good frequency for linearity testing.

Figure 133. Examples of (Left) Poor Frequency Choice vs (Right) Good Frequency Choice

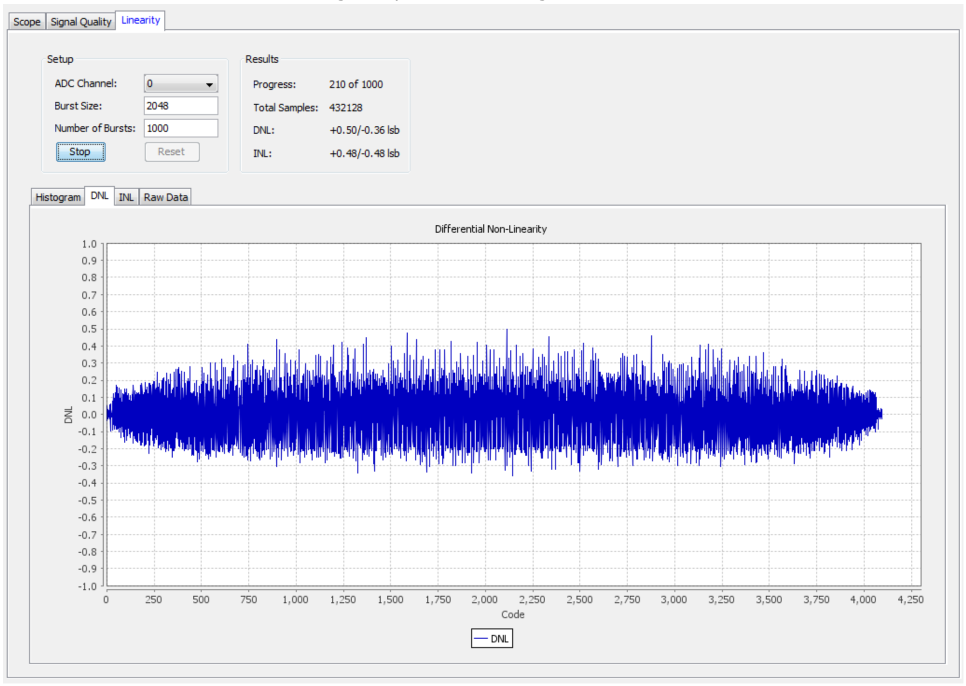


Differential Non-linearity View

Figure 134. Example of Good Differential Non-linearity

The **DNL** view shows the currently collected data. Ideally, you want your data to look like a straight line through the 0 on the x-axis. When there are not enough samples of data, the line appears rough. The line improves as more data is collected and averaged.

Each point in the graph represents how many LSB values a particular code differs from the ideal step size of 1 LSB. The **Results** box shows the highest positive and negative DNL values.



Setup

ADC Channel:	0
Burst Size:	2048
Number of Bursts:	1000

Results

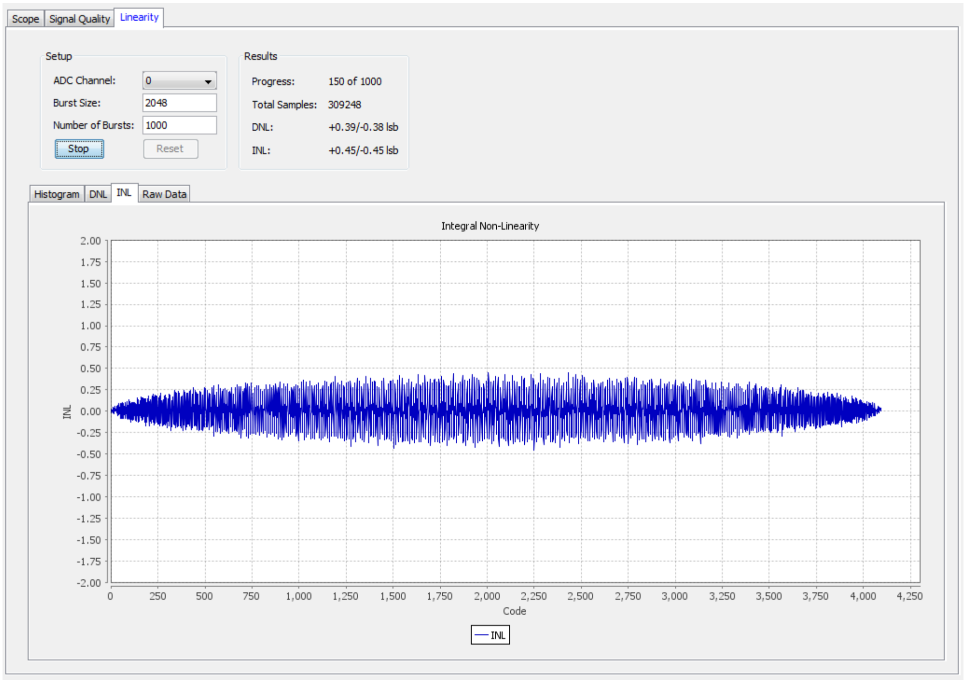
Progress:	210 of 1000
Total Samples:	432128
DNL:	+0.50/-0.36 lsb
INL:	+0.48/-0.48 lsb

Integral Non-linearity View

Figure 135. Example of Good Integral Non-linearity

The **INL** view shows currently collected data. Ideally, with a perfect ADC and enough samples, the graph appears as a straight line through 0 on the x-axis.

Each point in the graph represents how many LSB values a particular code differs from its expected point in the voltage slope. The **Results** box shows the highest positive and negative INL values.



ADC Channel: 0
Burst Size: 2048
Number of Bursts: 1000

Progress: 150 of 1000
Total Samples: 309248
DNL: +0.39/-0.38 lsb
INL: +0.45/-0.45 lsb

11.11 System Console Examples and Tutorials

Intel provides examples for performing board bring-up, creating a simple dashboard, and programming a Nios II processor. The `System_Console.zip` file contains design files for the board bring-up example. The Nios II Ethernet Standard `.zip` files contain the design files for the Nios II processor example.

Note: The instructions for these examples assume that you are familiar with the Intel Quartus Prime software, Tcl commands, and Platform Designer (Standard).

Related Links

On-Chip Debugging Design Examples Website
Contains the design files for the example designs that you can download.

11.11.1 Board Bring-Up with System Console Tutorial

You can perform low-level hardware debugging of Platform Designer (Standard) systems with System Console. You can debug systems that include IP cores instantiated in your Platform Designer (Standard) system or perform initial bring-up of your PCB. This board bring-up tutorial uses a Nios II Embedded Evaluation Kit (NEEK) board and USB cable. If you have a different development kit, you need to change the device and pin assignments to match your board and then recompile the design.

1. Setting Up the Board Bring-Up Design Example on page 262
2. Verifying Clock and Reset Signals on page 263
3. Verifying Memory and Other Peripheral Interfaces on page 263
4. Platform Designer (Standard) Modules for Board Bring-up Example on page 268

Related Links

Introduction to System Console on page 193

11.11.1.1 Setting Up the Board Bring-Up Design Example

To load the design example into the Intel Quartus Prime software and program your device, perform the following steps:

1. Unzip the `System_Console.zip` file to your local hard drive.
2. In the Intel Quartus Prime software, click **File ➤ Open Project** and select `Systemconsole_design_example.qpf`.
3. Change the device and pin assignments (LED, clock, and reset pins) in the `Systemconsole_design_example.qsf` file to match your board.
4. Click **Processing ➤ Start Compilation**
5. To program your device, follow these steps:
   a. Click **Tools ➤ Programmer**.
   b. Click **Hardware Setup**.
   c. Click the **Hardware Settings** tab.
   d. Under **Currently selected hardware**, click **USB-Blaster**, and click **Close**.

      Note: If you do not see the **USB-Blaster** option, then your device was not detected. Verify that the USB-Blaster driver is installed, your board is powered on, and the USB cable is intact.

      This design example uses a USB-Blaster cable. If you do not have a USB-Blaster cable and you are using a different cable type, then select your cable from the **Currently selected hardware** options.
   e. Click **Auto Detect**, and then select your device.
   f. Double-click your device under **File**.
   g. Browse to your project folder and click `Systemconsole_design_example.sof` in the subdirectory **output_files**.
   h. Turn on the **Program/Configure** option.
   i. Click **Start**.
   j. Close the Programmer.
6. Click **Tools** > **System Debugging Tools** > **System Console**.

Related Links

System_Console.zip file
Contains the design files for this tutorial.

11.11.1.2 Verifying Clock and Reset Signals

You can use the System Explorer pane to verify clock and reset signals.

Open the appropriate node and check for either a green clock icon or a red clock icon. You can use JTAG Debug command to verify clock and reset signals.

Related Links

- System Explorer Pane on page 198
- JTAG Debug Commands on page 211

11.11.1.3 Verifying Memory and Other Peripheral Interfaces

The Avalon-MM service accesses memory-mapped slaves via a suitable Avalon-MM master, which can be controlled by the host. You can use Tcl commands to read and write to memory with a master service.

11.11.1.3.1 Locating and Opening the Master Service

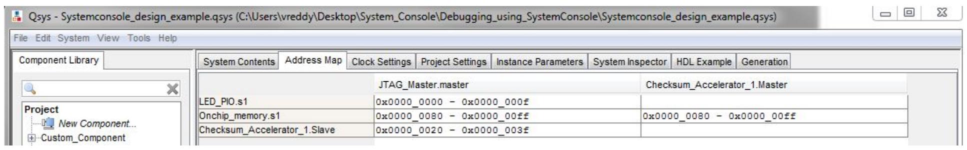
```
#Select the master service type and check for available service paths.
set service_paths [get_service_paths master]

#Set the master service path.
set master_service_path [lindex $service_paths 0]

#Open the master service.
set claim_path [claim_service master $master_service_path mylib]
```

11.11.1.3.2 Avalon-MM Slaves

The **Address Map** tab shows the address range for every Platform Designer (Standard) component. The Avalon-MM master communicates with slaves using these addresses.

The register maps for all Intel FPGA components are in their respective Data Sheets.

Figure 136. Address Map



Qsys - Systemconsole_design_example.qsys (C:\Users\vreddy\Desktop\System_Console\Debugging_using_SystemConsole\Systemconsole_design_example.qsys)

	JTAG_Master.master	Checksum_Accelerator_1.Master
LED_PIO.s1	0x0000_0000 – 0x0000_000f	
Onchip_memory.s1	0x0000_0080 – 0x0000_00ff	0x0000_0080 – 0x0000_00ff
Checksum_Accelerator_1.Slave	0x0000_0020 – 0x0000_003f	

Related Links

Data Sheets Website

Avalon-MM Commands

You can read or write the Avalon-MM interfaces using the master read and write commands. You can also use the master commands on slave services. If you are working on a slave service, the address field can be a register. (6)

Table 77. Avalon-MM Commands

Command	Arguments	Function
`master_write_memory`	*<service-path>* *<address>* *<list_of_byte_values>*	Writes the specified list of byte values to the specified service path and address.
`master_write_8`	*<service-path>* *<address>* *<list_of_byte_values>*	Writes the specified list of byte values to the specified service path and address, using 8-bit accesses.
`master_write_16`	*<service-path>* *<address>* *<list_of_16_bit_words>*	Writes the specified list of 16-bit values to the specified service path and address, using 16-bit accesses.
`master_write_from_file`	*<service-path>* *<file-name>* *<address>*	Writes the entire contents of the file to the specified service path and address. The file is treated as a binary file containing a stream of bytes.
`master_write_32`	*<service-path>* *<address>* *<list_of_32_bit_words>*	Writes the specified list of 32-bit values to the specified service path and address, using 32-bit accesses.
`master_read_memory`	[*-format <format>*] *<service-path>* *<address>* *<size_in_bytes>*	Returns a list of read values in bytes. Memory read starts at the specified base address. *Note:* The [*-format <format>*] is an optional argument. Specifying this argument makes this command accept data as 16 or 32-bit, instead as bytes. For example: `master_read_memory -format 16 <service_path> <addr> <count>`
`master_read_8`	*<service-path>* *<address>* *<size_in_bytes>*	Returns a list of *<size>* bytes. Read from memory starts at the specified base address, using 8-bit accesses.
`master_read_16`	*<service-path>* *<address>* *<size_in_multiples_of_16_bits>*	Returns a list of *<size>* 16-bit values. Read from memory starts at the specified base address, using 16-bit accesses.
`master_read_32`	*<service-path>* *<address>* *<size_in_multiples_of_32_bits>*	Returns a list of *<size>* 32-bit values. Read from memory starts at the specified base address, using 32-bit accesses.
`master_read_to_file`	*<service-path>* *<file-name>* *<address>* *<count>*	Reads the number of bytes specified by *<count>* from the memory address specified and creates (or overwrites) a file containing the values read. The file is written as a binary file.
`master_get_register_names`	*<service-path>*	When a register map is defined, returns a list of register names in the slave.

Note: Using the 8, 16, or 32 versions of the `master_read` or `master_write` commands is less efficient than using the `master_write_memory` or `master_read_memory` commands.

(6) Transfers performed in 16- and 32-bit sizes are packed in little-endian format.

11.11.1.3.3 Testing the PIO component

In this example design, the PIO connects to the LEDs of the board. Test if this component is operating properly and the LEDs are connected, by driving the outputs with the Avalon-MM master.

Table 78. Register Map for the PIO Core

Offset	Register Name		R/W	Fields				
				(n-1)	...	2	1	0
0	data	read access	R	Data value currently on PIO inputs.				
		write access	W	New value to drive on PIO outputs.				
1	direction		R/W	Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.				
2	interruptmask		R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.				
3	edgecapture		R/W	Edge detection for each input port.				

```
#Write the driver output values for the Parallel I/O component.
set offset 0x0; #Register address offset.
set value 0x7; #Only set bits 0, 1, and 2.
master_write_8 $claim_path $offset $value

#Read back the register value.
set offset 0x0
set count 0x1
master_read_8 $claim_path $offset $count

master_write_8 $claim_path 0x0 0x2; #Only set bit 1.

master_write_8 $claim_path 0x0 0xe; #Only set bits 1, 2, 3.

master_write_8 $claim_path 0x0 0x7; #Only set bits 0, 1, 2.

#Observe the LEDs turn on and off as you execute these Tcl commands.
#The LED is on if the register value is zero and off if the register value is
one.
#LED 0, LED 1, and LED 2 connect to the PIO.
#LED 3 connects to the interrupt signal of the CheckSum Accelerator.
```

11.11.1.3.4 Testing On-chip Memory

Test the memory with a recursive function that writes to incrementing memory addresses.

```
#Load the design example utility procedures for writing to memory.
source set_memory_values.tcl

#Write to the on-chip memory.
set base_address 0x80
set write_length 0x80
set value 0x5a5a5a5a
fill_memory $claim_path $base_address $write_length $value

#Verify the memory was written correctly.
#This utility proc returns 0 if the memory range is not uniform with this
value.
verify_memory $claim_path $base_address $write_length $value

#Check that the memory is re-initialized when reset.
#Trigger reset then observe verify_memory returns 0.
```

```
set jtag_debug_path [lindex [get_service_paths jtag_debug] 0]
set claim_jtag_debug_path [claim_service jtag_debug $jtag_debug_path mylib]
jtag_debug_reset_system $claim_jtag_debug_path; #Reset the connected on-chip
memory
#peripheral.
close_service jtag_debug $claim_jtag_debug_path
verify_memory $claim_path $base_address $write_length $value

#The on-chip memory component was parameterized to re-initialized to 0 on
reset.
#Check the actual value.
master_read_8 $claim_path 0x0 0x1
```

11.11.1.3.5 Testing the Checksum Accelerator

The Checksum Accelerator calculates the checksum of a data buffer in memory. It calculates the value for a specified memory buffer, sets the DONE bit in the status register, and asserts the interrupt signal. You should only read the result from the controller when both the DONE bit and the interrupt signal are asserted. The host should assert the interrupt enable control bit in order to check the interrupt signal.

Table 79. Register Map for Checksum Component

Offset (Bytes)	Hexadecimal value (after adding offset)	Register	Access	Bits (32 bits)							
				31-9	8	7-5	4	3	2	1	0
0	0x20	Status	Read/Write to clear							BUSY	DONE
4	0x24	Address	Read/Write	Read Address							
12	0x2C	Length	Read/Write	Length in bytes							
24	0x38	Control	Read/Write		Fixed Read Address Bit		Interrup t Enable	GO		INV	Clear
28	0x3C	Result	Read	Checksum result (upper 16 bits are zero)							

1.
```
#Pass the base address of the memory buffer Checksum Accelerator.
set base_address 0x20
set offset 4
set address_reg [expr {$base_address + $offset}]
set memory_address 0x80
master_write_32 $claim_path $address_reg $memory_address

#Pass the memory buffer to the Checksum Accelerator.
set length_reg [expr {$base_address + 12}]
set length 0x20
master_write_32 $claim_path $length_reg $length

#Write clear to status and control registers.
#Status register:
set status_reg $base_address
master_write_32 $claim_path $status_reg 0x0
#Control register:
set clear 0x1
set control_reg [expr {$base_address + 24}]
master_write_32 $claim_path $control_reg $clear

#Write GO to the control register.
set go 0x8
master_write_32 $claim_path $control_reg $go

#Cross check if the checksum DONE bit is set.
master_read_32 $claim_path $status_reg 0x1
```

```
#Is the DONE bit set?
#If yes, check the result and you are finished with the board bring-up
design example.
set result_reg [expr {$base_address + 28}]
master_read_16 $claim_path $result_reg 0x1
```

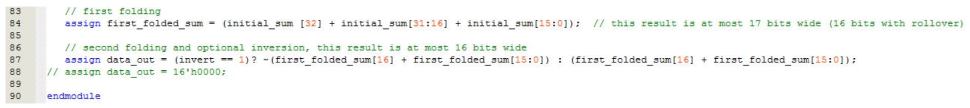
2. If the result is zero and the JTAG chain works properly, the clock and reset signals work properly, and the memory works properly, then the problem is the Checksum Accelerator component.

```
#Confirm if the DONE bit in the status register (bit 0)
#and interrupt signal are asserted.
#Status register:
master_read_32 $claim_path $status_reg 0x1
#Check DONE bit should return a one.

#Enable interrupt and go:
set interrupt_and_go 0x18
master_write_32 $claim_path $control_reg $interrupt_and_go
```

3. Check the control enable to see the interrupt signal. LED 3 (MSB) should be off. This indicates the interrupt signal is asserted.
4. You have narrowed down the problem to the data path. View the RTL to check the data path.
5. Open the `Checksum_transform.v` file from your project folder.
   - *<unzip dir>*`/System_Console/ip/checksum_accelerator/checksum_accelerator.v`
6. Notice that the `data_out` signal is grounded in Figure 137 on page 267 (uncommented line 87 and comment line 88). Fix the problem.
7. Save the file and regenerate the Platform Designer (Standard) system.
8. Re-compile the design and reprogram your device.
9. Redo the above steps, starting with Verifying Memory and Other Peripheral Interfaces on page 263 or run the Tcl script included with this design example.

```
source set_memory_and_run_checksum.tcl
```

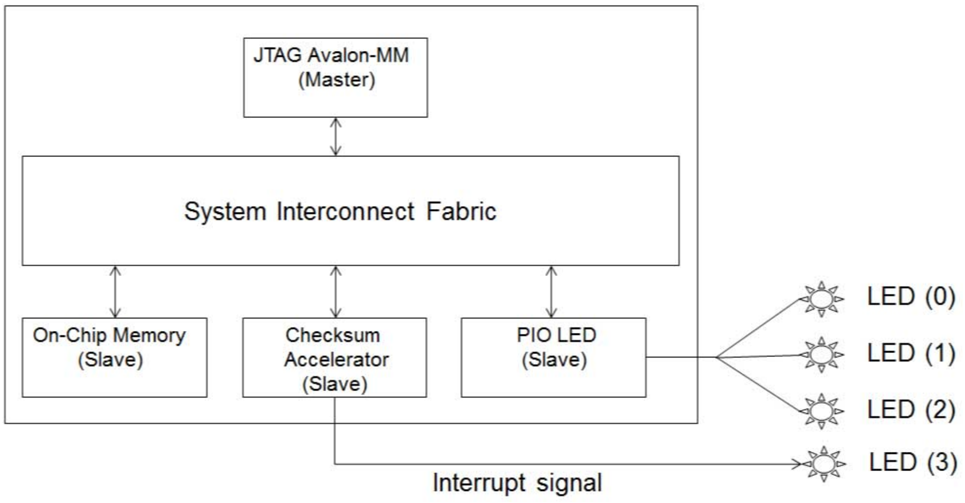
Figure 137. Checksum.v File



```
83      // first folding
84      assign first_folded_sum = (initial_sum [32] + initial_sum[31:16] + initial_sum[15:0]);  // this result is at most 17 bits wide (16 bits with rollover)
85
86      // second folding and optional inversion, this result is at most 16 bits wide
87      assign data_out = (invert == 1)? ~(first_folded_sum[16] + first_folded_sum[15:0]) : (first_folded_sum[16] + first_folded_sum[15:0]);
88   // assign data_out = 16'h0000;
89
90   endmodule
```

11.11.1.4 Platform Designer (Standard) Modules for Board Bring-up Example

Figure 138. Platform Designer (Standard) Modules for Board Bring-up Example



The Platform Designer (Standard) design for this example includes the following modules:

- JTAG to Avalon Master Bridge—Provides System Console host access to the memory-mapped IP in the design via the JTAG interface.
- On-chip memory—Simplest type of memory for use in an FPGA-based embedded system. The memory is implemented on the FPGA; consequently, external connections on the circuit board are not necessary.
- Parallel I/O (PIO) module—Provides a memory-mapped interface for sampling and driving general I/O ports.
- Checksum Accelerator—Calculates the checksum of a data buffer in memory. The Checksum Accelerator consists of the following:
  - Checksum Calculator (`checksum_transform.v`)
  - Read Master (`slave.v`)
  - Checksum Controller (`latency_aware_read_master.v`)

11.11.1.4.1 Checksum Accelerator Functionality

The base address of the memory buffer and data length passes to the Checksum Controller from a memory-mapped master. The Read Master continuously reads data from memory and passes the data to the Checksum Calculator. When the checksum calculations finish, the Checksum Calculator issues a valid signal along with the checksum result to the Checksum Controller. The Checksum Controller sets the DONE bit in the status register and also asserts the interrupt signal. You should only read the result from the Checksum Controller when the DONE bit and interrupt signal are asserted.

11.11.2 Nios II Processor Example

This example programs the Nios II processor on your board to run the count binary software example included in the Nios II installation. This is a simple program that uses an 8-bit variable to repeatedly count from 0x00 to 0xFF. The output of this variable is displayed on the LEDs on your board. After programming the Nios II processor, you use System Console processor commands to start and stop the processor.

To run this example, perform the following steps:

1. Download the Nios II Ethernet Standard Design Example for your board from the Altera website.
2. Create a folder to extract the design. For this example, use `C:\Count_binary`.
3. Unzip the Nios II Ethernet Standard Design Example into `C:\Count_binary`.
4. In a Nios II command shell, change to the directory of your new project.
5. Program your board. In a Nios II command shell, type the following:

   ```
   nios2-configure-sof niosii_ethernet_standard_<board_version>.sof
   ```

6. Using Nios II Software Build Tools for Eclipse, create a new Nios II Application and BSP from Template using the **Count Binary** template and targeting the Nios II Ethernet Standard Design Example.
7. To build the executable and linkable format (ELF) file (`.elf`) for this application, right-click the **Count Binary** project and select **Build Project**.
8. Download the `.elf` file to your board by right-clicking **Count Binary** project and selecting **Run As, Nios II Hardware**.
   - The LEDs on your board provide a new light show.
9. Type the following:

   ```
   system-console; #Start System Console.

   #Set the processor service path to the Nios II processor.
   set niosii_proc [lindex [get_service_paths processor] 0]

   set claimed_proc [claim_service processor $niosii_proc mylib]; #Open the
   service.

   processor_stop $claimed_proc; #Stop the processor.
   #The LEDs on your board freeze.

   processor_run $claimed_proc; #Start the processor.
   #The LEDs on your board resume their previous activity.

   processor_stop $claimed_proc; #Stop the processor.

   close_service processor $claimed_proc; #Close the service.
   ```

   - The `processor_step`, `processor_set_register`, and `processor_get_register` commands provide additional control over the Nios II processor.

Related Links

- Nios II Ethernet Standard Design Example
- Nios II Gen2 Software Developer's Handbook

11.11.2.1 Processor Commands

Table 80. Processor Commands

Command [7]	Arguments	Function
`processor_download_elf`	*<service-path>* *<elf-file-path>*	Downloads the given Executable and Linking Format File (`.elf`) to memory using the master service associated with the processor. Sets the processor's program counter to the `.elf` entry point.
`processor_in_debug_mode`	*<service-path>*	Returns a non-zero value if the processor is in debug mode.
`processor_reset`	*<service-path>*	Resets the processor and places it in debug mode.
`processor_run`	*<service-path>*	Puts the processor into run mode.
`processor_stop`	*<service-path>*	Puts the processor into debug mode.
`processor_step`	*<service-path>*	Executes one assembly instruction.
`processor_get_register_names`	*<service-path>*	Returns a list with the names of all of the processor's accessible registers.
`processor_get_register`	*<service-path>* *<register_name>*	Returns the value of the specified register.
`processor_set_register`	*<service-path>* *<register_name>* *<value>*	Sets the value of the specified register.

Related Links

Nios II Processor Example on page 269

11.12 On-Board Intel FPGA Download Cable II Support

System Console supports an On-Board Intel FPGA Download Cable II circuit via the USB Debug Master IP component. This IP core supports the master service.

Not all Stratix V boards support the On-Board Intel FPGA Download Cable II. For example, the transceiver signal integrity board does not support the On-Board Intel FPGA Download Cable II.

11.13 About Using MATLAB and Simulink in a System Verification Flow

You can test system development in System Console using MATLAB and Simulink, and set up a system verification flow using the Intel FPGA Hardware in the Loop (HIL) tools. In this approach, you deploy the design hardware to run in real time, and simulate your system's surrounding components in a software environment. The HIL approach allows you to use the flexibility of software tools with the real-world accuracy and speed of hardware. You can gradually introduce more hardware components to your system verification testbench. This technique gives you more

[7] If your system includes a Nios II/f core with a data cache, it may complicate the debugging process. If you suspect the Nios II/f core writes to memory from the data cache at nondeterministic intervals; thereby, overwriting data written by the System Console, you can disable the cache of the Nios II/f core while debugging.

control over the integration process as you tune and validate your system. When your full system is integrated, the HIL approach allows you to provide stimuli via software to test your system under a variety of scenarios.

Advantages of HIL Approach

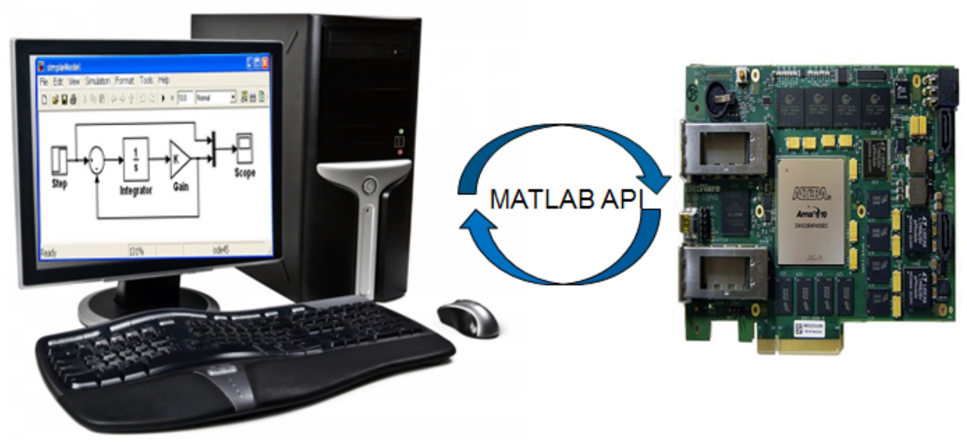
- Avoid long computational delays for algorithms with high processing rates
- API helps to control, debug, visualize, and verify FPGA designs all within the MATLAB environment
- FPGA results are read back by the MATLAB software for further analysis and display

Required Tools and Components

- MATLAB software
- DSP Builder for Intel FPGAs software
- Intel Quartus Prime software
- Intel FPGA

Note: The DSP Builder for Intel FPGAs installation bundle includes the System Console MATLAB API.

Figure 139. Hardware in the Loop Host-Target Setup



Supported MATLAB API Commands

You can perform your work from the MATLAB environment and leverage the capability of System Console to read and write to masters and slaves. By using the supported MATLAB API commands, you do not have to launch the System Console software. The supported commands are the following:

- `SystemConsole.refreshMasters;`
- `M = SystemConsole.openMaster(1);`
- `M.write (type, byte address, data);`
- `M.read (type, byte address, number of words);`
- `M.close`

Example 33. MATLAB API Script Example

```
SystemConsole.refreshMasters; %Investigate available targets
M = SystemConsole.openMaster(1); %Creates connection with FPGA target
%%%%%%%% User Application %%%%%%%%%%%%
....
M.write('uint32',write_address,data); %Send data to FPGA target
....
data = M.read('uint32',read_address,size); %Read data from FPGA target
....
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
M.close; %Terminates connection to FPGA target
```

High-Level Flow

1. Install the DSP Builder for Intel FPGAs software so you have the necessary libraries to enable this flow
2. Build your design using Simulink and the DSP Builder for Intel FPGAs libraries (DSP Builder for Intel FPGAs helps to convert the Simulink design to HDL)
3. Include Avalon-MM components in your design (DSP Builder for Intel FPGAs can port non-Avalon-MM components)
4. Include Signals and Control blocks in your design
5. Use boundary blocks to separate synthesizable and non-synthesizable logic
6. Integrate your DSP system in Platform Designer (Standard)
7. Program your Intel FPGA
8. Use the supported MATLAB API commands to interact with your Intel FPGA

Related Links

Hardware in the Loop from the MATLAB Simulink Environment white paper

11.14 Deprecated Commands

The table lists commands that have been deprecated. These commands are currently supported, but are targeted for removal from System Console.

Note: All `dashboard_`*<name>* commands are deprecated and replaced with `toolkit_`*<name>* commands for Intel Quartus Prime software15.1, and later.

Table 81. Deprecated Commands

Command	Arguments	Function
`open_service`	*<service_type>* *<service_path>*	Opens the specified service type at the specified path. Calls to `open_service` may be replaced with calls to `claim_service` providing that the return value from `claim_service` is stored and used to access and close the service.

11.15 Document Revision History

Table 82. Document Revision History

Date	Version	Changes
2017.05.08	17.0.0	• Created topic *Convert your Dashboard Scripts to Toolkit API*. • Removed *Registering the Service* Example from *Toolkit API Script Examples*, and added corresponding code snippet to *Registering a Toolkit*. • Moved *.toolkit Description File Example* under *Creating a Toolkit Description File*. • Renamed *Toolkit API GUI Example .toolkit File* to *.toolkit Description File Example*. • Updated examples on Toolkit API to reflect current supported syntax.
2015.11.02	15.1.0	• Edits to Toolkit API content and command format. • Added Toolkit API design example. • Added graphic to *Introduction to System Console*. • Deprecated Dashboard. • Changed instances of *Quartus II* to *Intel Quartus Prime*.
October 2015	15.1.0	• Added content for Toolkit API — Required .toolkit and Tcl files — Registering and launching the toolkit — Toolkit discovery and matching toolkits to IP — Toolkit API commands table
May 2015	15.0.0	Added information about how to download and start System Console stand-alone.
December 2014	14.1.0	• Added overview and procedures for using ADC Toolkit on MAX 10 devices. • Added overview for using MATLAB/Simulink Environment with System Console for system verification.
June 2014	14.0.0	Updated design examples for the following: board bring-up, dashboard service, Nios II processor, design service, device service, monitor service, bytestream service, SLD service, and ISSP service.
November 2013	13.1.0	Re-organization of sections. Added high-level information with block diagram, workflow, SLD overview, use cases, and example Tcl scripts.
June 2013	13.0.0	Updated Tcl command tables. Added board bring-up design example. Removed SOPC Builder content.
November 2012	12.1.0	Re-organization of content.
August 2012	12.0.1	Moved Transceiver Toolkit commands to Transceiver Toolkit chapter.
June 2012	12.0.0	Maintenance release. This chapter adds new System Console features.
November 2011	11.1.0	Maintenance release. This chapter adds new System Console features.
May 2011	11.0.0	Maintenance release. This chapter adds new System Console features.
December 2010	10.1.0	Maintenance release. This chapter adds new commands and references for Qsys.
July 2010	10.0.0	Initial release. Previously released as the System Console User Guide, which is being obsoleted. This new chapter adds new commands.

Related Links

Documentation Archive
For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

12 Debugging Transceiver Links

The Transceiver Toolkit helps you optimize high-speed serial links in your board design by providing real-time control, monitoring, and debugging of the transceiver links running on your board.

The Transceiver Toolkit allows you to:

- Control the transmitter or receiver channels to optimize transceiver settings and hardware features.
- Test bit-error rate (BER) while running multiple links at the target data rate.
- Run auto sweep tests to identify the best physical media attachment (PMA) settings for each link.
- For Stratix V devices, view the receiver horizontal and vertical eye margin during testing.
- Test multiple devices across multiple boards simultaneously.

Note: The Transceiver Toolkit runs from the System Console framework.

To launch the toolkit, click **Tools ➤ System Debugging Tools ➤ Transceiver Toolkit**. Alternatively, you can run Tcl scripts from the command-line:

```
system-console --script=<name of script>
```

For an online demonstration using the Transceiver Toolkit to run a high-speed link test with one of the design examples, refer to the Transceiver Toolkit Online Demo on the Altera website.

Related Links

- On-Chip Debugging Design Examples
- Transceiver Toolkit Online Demo
- Transceiver Toolkit for Intel Arria 10 Devices (OTCVRKITA10)
  26 Minutes Online Course
- Transceiver Toolkit for 28-nm Devices (OTCVR1100)
  39 Minutes Online Course

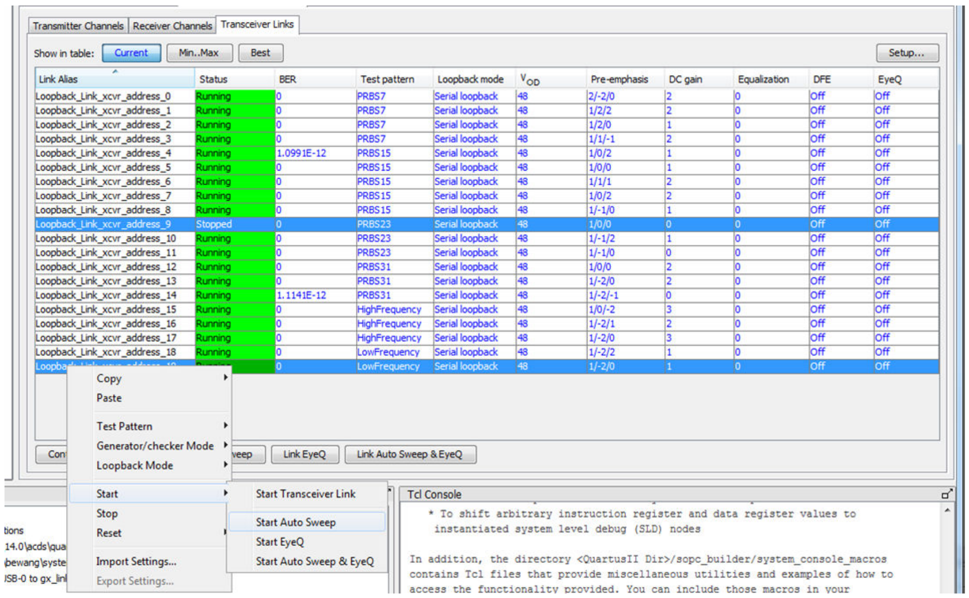
12.1 Channel Manager

The Channel Manager is the graphical component of the Transceiver Toolkit. The Channel Manager allows you to configure and control transceiver channels and links, and adjust programmable analog settings to improve the signal integrity of the link. The Channel Manager is in the Workspace area of the System Console.

The Channel Manager consists of three tabs:

- **Transmitter Channels**
- **Receiver Channels**
- **Transceiver Links**

Figure 140. Transceiver Links Tab of the Channel Manager



Link Alias	Status	BER	Test pattern	Loopback mode	V_{OD}	Pre-emphasis	DC gain	Equalization	DFE	EyeQ
Loopback_Link_xcvr_address_0	Running	0	PRBS7	Serial loopback	48	2/-2/0	2	0	Off	Off
Loopback_Link_xcvr_address_1	Running	0	PRBS7	Serial loopback	48	1/2/2	2	0	Off	Off
Loopback_Link_xcvr_address_2	Running	0	PRBS7	Serial loopback	48	1/2/0	1	0	Off	Off
Loopback_Link_xcvr_address_3	Running	0	PRBS7	Serial loopback	48	1/1/-1	2	0	Off	Off
Loopback_Link_xcvr_address_4	Running	1.0991E-12	PRBS15	Serial loopback	48	1/0/2	1	0	Off	Off
Loopback_Link_xcvr_address_5	Running	0	PRBS15	Serial loopback	48	1/0/0	1	0	Off	Off
Loopback_Link_xcvr_address_6	Running	0	PRBS15	Serial loopback	48	1/1/1	2	0	Off	Off
Loopback_Link_xcvr_address_7	Running	0	PRBS15	Serial loopback	48	1/0/2	2	0	Off	Off
Loopback_Link_xcvr_address_8	Running	0	PRBS15	Serial loopback	48	1/-1/0	1	0	Off	Off
Loopback_Link_xcvr_address_9	Stopped	0	PRBS23	Serial loopback	48	1/0/0	0	0	Off	Off
Loopback_Link_xcvr_address_10	Running	0	PRBS23	Serial loopback	48	1/-1/2	1	0	Off	Off
Loopback_Link_xcvr_address_11	Running	0	PRBS23	Serial loopback	48	1/-1/0	0	0	Off	Off
Loopback_Link_xcvr_address_12	Running	0	PRBS31	Serial loopback	48	1/0/0	2	0	Off	Off
Loopback_Link_xcvr_address_13	Running	0	PRBS31	Serial loopback	48	1/-2/0	2	0	Off	Off
Loopback_Link_xcvr_address_14	Running	1.1141E-12	PRBS31	Serial loopback	48	1/-2/-1	0	0	Off	Off
Loopback_Link_xcvr_address_15	Running	0	HighFrequency	Serial loopback	48	1/0/-2	3	0	Off	Off
Loopback_Link_xcvr_address_16	Running	0	HighFrequency	Serial loopback	48	1/-2/1	2	0	Off	Off
Loopback_Link_xcvr_address_17	Running	0	HighFrequency	Serial loopback	48	1/-2/0	3	0	Off	Off
Loopback_Link_xcvr_address_18	Running	0	LowFrequency	Serial loopback	48	1/-2/2	1	0	Off	Off
Loopba[illegible]	[illegible]	0	LowFrequency	Serial loopback	48	1/-2/0	1	0	Off	Off

```
  * To shift arbitrary instruction register and data register values to
    instantiated system level debug (SLD) nodes

In addition, the directory <QuartusII Dir>/sopc_builder/system_console_macros
contains Tcl files that provide miscellaneous utilities and examples of how to
access the functionality provided. You can include those macros in your
```

Channel Manager Functions

The Channel Manager simplifies actions such as:

- Copying and pasting settings—Copy, paste, import, and export PMA settings to and from channels.
- Importing and exporting settings— To export PMA settings to a text file, select a row in the Channel Manager. To apply the PMA settings from a text file, select one or more rows in the Channel Manager. The PMA settings in the text file apply to a single channel. When you import the PMA settings from a text file, you are duplicating one set of PMA settings for all the channels you select.
- Starting and stopping tests—The Channel Manager allows you to start and stop tests by right-clicking the channels. You can select two or more rows in the Channel Manager to start or stop test for multiple channels.

Related Links

- System Explorer Pane on page 198
- System Console GUI on page 197
- User Interface Settings Reference on page 300

12.1.1 Channel Display Modes

The three channel display modes are:

- **Current** (default)—shows the current values from the device. Blue text indicates that the settings are live.
- **Min/Max**—shows the minimum and maximum values to be used in the auto sweep.
- **Best**—shows the best tested values from the last completed auto sweep run.

Note: The **Transmitter Channels** tab only shows the **Current** display mode. The Transceiver Toolkit requires both a transmitter channel and a receiver channel to perform Auto sweep tests.

12.2 Transceiver Debugging Flow Walkthrough

These steps describe the high-level process of debugging transceivers in your design.

1. Configuring your System with Debugging Components on page 276.
2. Programming the Design into an Intel FPGA on page 287.
3. Loading the Design in the Transceiver Toolkit on page 288.
4. Linking Hardware Resources on page 288.
5. Verifying Hardware Connections on page 292.
6. Identifying Transceiver Channels on page 292.
7. Running Link Tests on page 293 or Controlling PMA Analog Settings on page 297.

12.3 Configuring your System with Debugging Components

The configuration of the debugging system varies by device family.

12.3.1 Adapting an Intel FPGA Design Example

Design examples allow you to quickly test the functionality of the receiver and transmitter channels in your design. You can modify and customize the design examples to match your intended transceiver design and signal integrity development board.

1. Download a design example from the On-Chip Debugging Design Examples page of the Intel FPGA website.
2. Open the Intel Quartus Prime and click **Project ➤ Restore Archived Project** to restore the design example project archive.
3. Compare the development board and device specified in the `readme.txt` file with your board and device:

Option	Description
Same development board and same device	Directly program the device with the programming file included in the example.
Same board, different device	Choose the appropriate device and recompile the design.

Option	Description
Different board	Edit the necessary pin assignments and recompile the design example.

4. To recompile the design, you must make your modifications to the system configuration in Platform Designer (Standard), regenerate in Platform Designer (Standard), and recompile the design in the Intel Quartus Prime software to generate a new programming file.

Once you recompile your design, you can:

- Change the transceiver settings in the design examples and observe the effects on transceiver link performance
- Isolate and verify the high-speed serial links without debugging other logic in your design.

Refer to the `readme.txt` of each design example for more information.

12.3.1.1 Modifying Stratix V Design Examples

You can adapt Intel FPGA design examples to experiment with configurations that match your own design. For example, you can change data rate, number of lanes, PCS-PMA width, FPGA-fabric interface width, or input reference clock frequency. To modify the design examples, change the IP core parameters and regenerate the system in Platform Designer (Standard). Next, update the top-level design file, and re-assign device I/O pins as necessary.

To modify a Stratix V design example PHY block to match your design, follow these steps:

1. Determine the number of channels your design requires.
2. Open the <*project name*>`.qpf` for the design example in the Intel Quartus Prime software.
3. Click **Tools ➤ Platform Designer (Standard)**.
4. On the **System Contents** tab, right-click the PHY block and click **Edit**. Specify options for the PHY block to match your design requirement for number of lanes, data rate, PCS-PMA width, FPGA-fabric interface width, and input reference clock frequency.
5. Specify a multiple of the FPGA-fabric interface data width for **Avalon Data Symbol Size**. The available values are **8** or **10**. Click **Finish**.
6. Delete any timing adapter from the design. The timing adaptors are not required.
7. From the IP Catalog, add one **Data Pattern Generator** and **Data Pattern Checker** for each transmitter and receiver lane.
8. Right-click **Data Pattern Generator** and click **Edit**. Specify a value for **ST_DATA_W** that matches the FPGA-fabric interface width.
9. Right-click **Data Pattern Checker** and click **Edit**. Specify a value for **ST_DATA_W** that matches the FPGA-fabric interface width.
10. From the IP Catalog, add a **Transceiver Reconfiguration Controller**.
11. Right-click **Transceiver Reconfiguration Controller** and click **Edit**. Specify 2* number of lanes for the number of reconfigurations interfaces. Click **finish**.

12. Create connections for the data pattern generator and data pattern checker components. Right-click the net name in the **System Contents** tab and specify the following connections.

From		To	
Block Name	**Net Name**	**Block Name**	**Net Name**
clk_100	clk	data_pattern_generator	csr_clk
clk_100	clk_reset	data_pattern_generator	csr_clk_reset
master_0	master	data_pattern_generator	csr_slave
xcvr_*_phy_0	tx_clk_out0	data_pattern_generator	pattern_out_clk
xcvr_*_phy_0	tx_parallel_data0	data_pattern_generator	pattern_out
clk_100	clk	data_pattern_checker	csr_clk
clk_100	clk_reset	data_pattern_checker	csr_clk_reset
master_0	master	data_pattern_checker	csr_slave
xcvr_*_phy_0	rx_clk_out0	data_pattern_checker	pattern_in_clk
xcvr_*_phy_0	rx_parallel_data0	data_pattern_checker	pattern_in

13. Click **System ➤ Assign Base Addresses**.
14. Connect the reset port of timing adapters to `clk_reset` of `clk_100`.
15. To implement the changes to the system, click **Generate ➤ Generate HDL**.
16. If you modify the number of lanes in the PHY, you must update the top-level file accordingly. The following example shows Verilog HDL code for a two-channel design that declares input and output ports in the top-level design. The example design includes the low latency PHY IP core. If you modify the PHY parameters, you must modify the top-level design with the correct port names. Platform Designer (Standard) displays an example of the PHY, click **Generate ➤ HDL Example**.

```
module low_latency_10g_1ch DUT (
                        input  wire GXB_RXL11,
                        input  wire GXB_RXL12,
                        output wire GXB_TXL11,
                        output wire GXB_TX12
                        );
                        .....
                        low_latency_10g_1ch DUT (
                        .....
                        .xcvr_low_latency_phy_0_tx_serial_data_export
({GXB_TXL11,     GXB_TXL12}),
                        .xcvr_low_latency_phy_0_rx_serial_data_export
({GXB_RXL11,         GXB_TXL12}),
                        .....
                        );
```

17. From the Intel Quartus Prime software, click **Assignments ➤ Pin Planner** and update pin assignments to match your board.
18. Edit the design's Synopsys Design Constraints (`.sdc`) to reflect the reference clock change. Ignore the reset warning messages.
19. Click **Start ➤ Start Compilation** to recompile the design.

12.3.1.1.1 Generating reconfig_clk from an Internal PLL

You can use an internal PLL to generate the `reconfig_clk`, by changing the Platform Designer (Standard) connections to delay offset cancellation until the generated clock is stable.

- If there is no free running clock within the required frequency range of the reconfiguration clock, add a PLL to the top-level of the design example. The frequency range varies depending on the device family. Refer to the device family data sheet for your device.
- When using an internal PLL, hold off offset cancellation until the generated clock is stable. You do this by connecting the `pll_locked` signal of the internal PLL to the `.clk_clk_in_reset_n` port of the Platform Designer (Standard) system, instead of the `system_reset` signal.
- Implement the filter logic, inverter, and synchronization to the `reconfig_clk` outside of the Platform Designer (Standard) system with your own logic.

You can find the support solution in the Intel FPGA Knowledge Base. The solution applies to only Arria V, Cyclone V, Stratix IV GX/GT, and Stratix V devices.

12.3.2 Stratix V Debug System Configuration

For Stratix V designs, the Transceiver Toolkit configuration requires instantiation of the JTAG to Avalon Bridge and Reconfiguration Controller IP cores. Click **Tools ➤ IP Catalog** to parameterize, generate, and instantiate the following debugging components for Stratix V designs.

Table 83. Stratix V / 28nm Transceiver Toolkit IP Core Configuration

Component	Debugging Functions	Parameterization Notes	Connect To
Transceiver Native PHY	Supports all debugging functions	• If **Enable 10G PCS** is enabled, **10G PCS protocol mode** must be set to **basic** on the **10G PCS** tab.	• Avalon-ST Data Pattern Checker • Avalon-ST Data Pattern Generator • JTAG to Avalon Master Bridge • Reconfiguration controller
Custom PHY	Test all possible transceiver parallel data widths	• Set lanes, group size, serialization factor, data rate, and input clock frequency to match your application. • Turn on **Avalon data interfaces**. • Disable **8B/10B**. • Set **Word alignment mode** to **manual**. • Disable **rate match FIFO**. • Disable **byte ordering block**.	• Avalon-ST Data Pattern Checker • Avalon-ST Data Pattern Generator • JTAG to Avalon Master Bridge • Reconfiguration controller
Low Latency PHY	Test at more than 8.5 Gbps in GT devices or use of PMA direct mode (such as when using six channels in one quad)	• Set **Phase compensation FIFO mode** to **EMBEDDED** above certain data rates and set to **NONE** for PMA direct mode. • Turn on **Avalon data interfaces**. • Set serial loopback mode to enable serial loopback controls in the toolkit.	• Avalon-ST Data Pattern Checker • Avalon-ST Data Pattern Generator • JTAG to Avalon Master Bridge • Reconfiguration controller

continued...

Component	Debugging Functions	Parameterization Notes	Connect To
Intel-Avalon Data Pattern Generator	Generates standard data test patterns at Avalon-ST source ports	• Select **PRBS7**, **PRBS15**, **PRBS23**, **PRBS31**, **high frequency**, or **low frequency** patterns. • Turn on **Enable Bypass interface** for connection to design logic.	• PHY input port • JTAG to Avalon Master Bridge • Your design logic
Intel-Avalon Data Pattern Checker	Validates incoming data stream against test patterns accepted on Avalon streaming sink ports	Specify a value for ST_DATA_W that matches the FPGA-fabric interface width.	• PHY output port • JTAG to Avalon Master Bridge
Reconfiguration Controller	Supports PMA control and other transceiver settings	• Connect the reconfiguration controller to • Connect `reconfig_from_xcvr` to `reconfig_to_xcvr`. • Enable Analog controls. • Turn on **Enable Eye Viewerblock** to enable signal eye analysis (Stratix V only) • Turn on **Enable Bit Error Rate Block** for BER testing • Turn on **Enable decision feedback equalizer (DFE) block** for link optimization • Enable DFE block	• PHY input port • JTAG to Avalon Master Bridge
JTAG to Avalon Master Bridge	Accepts encoded streams of bytes with transaction data and initiates Avalon-MM transactions	N/A	• PHY input port • Avalon-ST Data Pattern Checker • Avalon-ST Data Pattern Generator • Reconfiguration Controller

12.3.2.1 Bit Error Rate Test Configuration (Stratix V)

Use the following configuration to perform bit rate error testing in Stratix V designs.

Figure 141. Bit Error Rate Test Configuration (Stratix V)

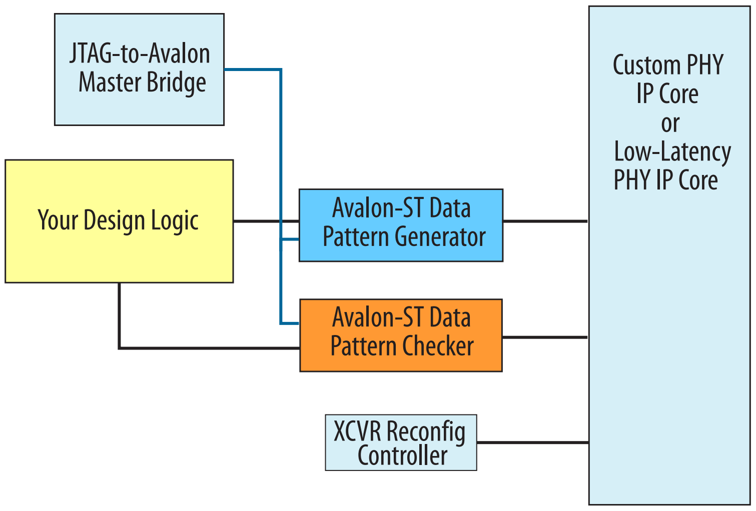

JTAG-to-Avalon Master Bridge
Your Design Logic
Avalon-ST Data Pattern Generator
Avalon-ST Data Pattern Checker
XCVR Reconfig Controller
Custom PHY IP Core or Low-Latency PHY IP Core

Table 84. System Connections: Bit Error Rate Tests

From	To
Your Design Logic	Data Pattern Generator bypass port
Data Pattern Generator	PHY input port
JTAG to Avalon Master Bridge	Intel FPGA Avalon Data Pattern Generator
JTAG to Avalon Master Bridge	Intel FPGA Avalon Data Pattern Checker
JTAG to Avalon Master Bridge	PHY input port
Data Pattern Checker	PHY output port
Transceiver Reconfiguration Controller	PHY input port

Related Links

Running BER Tests on page 294

12.3.2.2 PRBS Signal Eye Test Configuration (Stratix V)

Use the following configuration to perform PRBS signal eye testing in Stratix V designs.

Figure 142. PRBS Signal Eye Test Configuration (Stratix V)

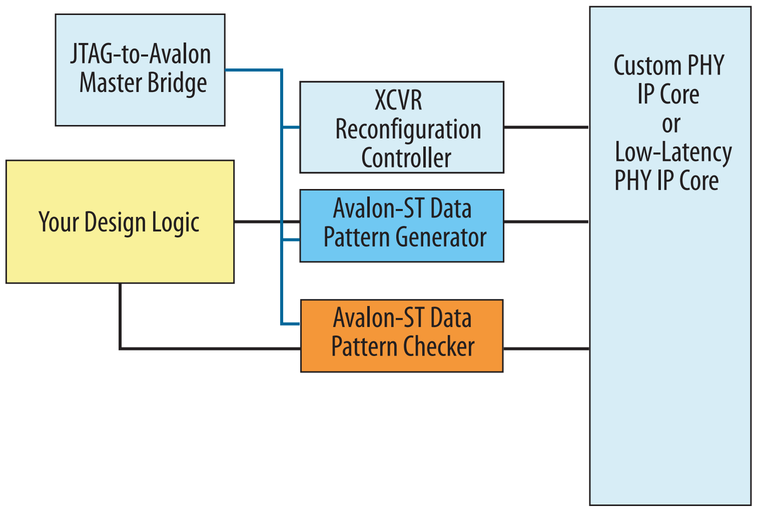

JTAG-to-Avalon Master Bridge
XCVR Reconfiguration Controller
Your Design Logic
Avalon-ST Data Pattern Generator
Avalon-ST Data Pattern Checker
Custom PHY IP Core or Low-Latency PHY IP Core

Table 85. System Connections: PRBS Signal Eye Tests (Stratix V)

From	To
Your Design Logic	Data Pattern Generator bypass port
Data Pattern Generator	PHY input port
JTAG to Avalon Master Bridge	Intel Avalon Data Pattern Generator
JTAG to Avalon Master Bridge	Intel Avalon Data Pattern Checker
Data Pattern Checker	PHY output port

continued...

From	To
JTAG to Avalon Master Bridge	Transceiver Reconfiguration Controller
JTAG to Avalon Master Bridge	PHY input port
Transceiver Reconfiguration Controller	PHY input port

Related Links

Running PRBS Signal Eye Tests (Stratix V only) on page 295

12.3.2.2.1 Enabling Serial Bit Comparator Mode (Stratix V)

Serial bit comparator mode allows you to run Eye Viewer diagnostic features with any PRBS patterns or user-design data, without disrupting the data path. For Stratix V devices, you must enable **Serial bit comparator** mode.

To enable this mode for Stratix V devices, you must enable the following debugging component options when configuring the debugging system:

Table 86. Component Settings for Serial Bit Comparator Mode

Debugging Component	Setting for Serial Bit Mode[8]
Transceiver Reconfiguration Controller	Turn on **Enable Eye Viewer block** and **Enable Bit Error Rate Block**
Data Pattern Generator[9]	Turn on **Enable Bypass interface**

Serial bit comparator mode is less accurate than **Data pattern checker** mode for single bit error checking. Do not use **Serial bit comparator** mode if you require an exact error rate. Use the **Serial bit comparator** mode for checking a large window of error. The toolkit does not read the bit error counter in real-time because it reads through the memory-mapped interface. Serial bit comparator mode has the following hardware limitations for Stratix V devices:

- Toolkit uses serial bit checker only on a single channel per reconfiguration controller at a time.
- When the serial bit checker is running on channel *n*, you can change only the V_{OD}, pre-emphasis, DC gain, and Eye Viewer settings on that channel. Changing or enabling DFE or CTLE can cause corruption of the serial bit checker results.
- When the serial bit checker is running on a channel, you cannot change settings on any other channel on the same reconfiguration controller.
- When the serial bit checker is running on a channel, you cannot open any other channel in the Transceiver Toolkit.
- When the serial bit checker is running on a channel, you cannot copy PMA settings from any channel on the same reconfiguration controller.

12.3.2.3 Custom Traffic Signal Eye Test Configuration (Stratix V)

Use the following configuration to perform custom traffic signal eye testing in Stratix V designs.

[8] Settings in Table 86 on page 282 are supported in Stratix V devices only.

[9] Limited support for Data Pattern Generator or data pattern in Serial Bit Mode.

Figure 143. System Configuration: Custom Traffic Signal Eye Tests (Stratix V)

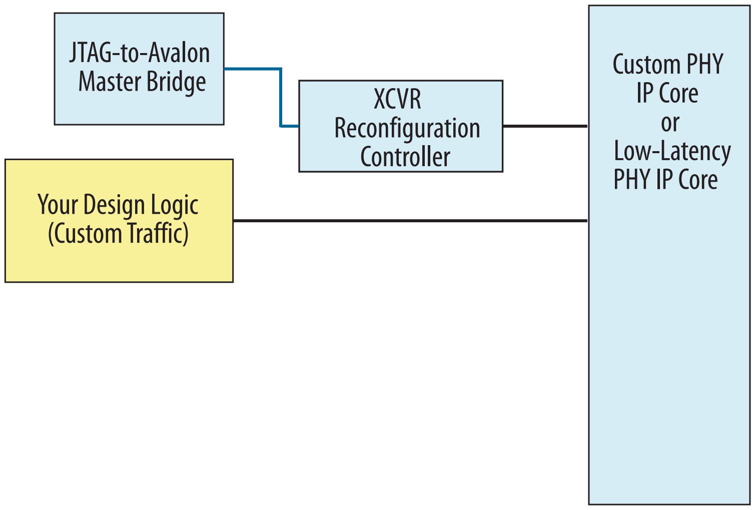

JTAG-to-Avalon Master Bridge
XCVR Reconfiguration Controller
Custom PHY IP Core or Low-Latency PHY IP Core
Your Design Logic (Custom Traffic)

Table 87. System Connections: Custom Traffic Signal Eye Tests (Stratix V)

From	To
Your design logic with custom traffic	PHY input port
JTAG to Avalon Master Bridge	Transceiver Reconfiguration Controller
JTAG to Avalon Master Bridge	PHY input port
Transceiver Reconfiguration Controller	PHY input port

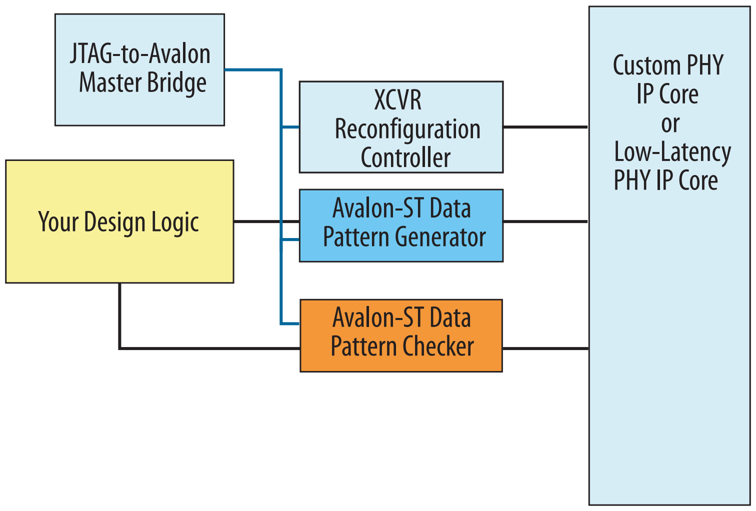
Related Links

Running Custom Traffic Tests (Stratix V only) on page 296

12.3.2.4 Link Optimization Test Configuration (Stratix V)

Use the following configuration for link optimization tests in Stratix V devices.

Figure 144. System Configuration: Link Optimization Tests (Stratix V)


JTAG-to-Avalon Master Bridge
XCVR Reconfiguration Controller
Custom PHY IP Core or Low-Latency PHY IP Core
Your Design Logic
Avalon-ST Data Pattern Generator
Avalon-ST Data Pattern Checker

From	To
Your Design Logic	Data Pattern Generator bypass port
Data Pattern Generator	PHY input port
JTAG to Avalon Master Bridge	Altera Avalon Data Pattern Generator
JTAG to Avalon Master Bridge	Altera Avalon Data Pattern Checker
Data Pattern Checker	PHY output port
JTAG to Avalon Master Bridge	Transceiver Reconfiguration Controller
JTAG to Avalon Master Bridge	PHY input port
Transceiver Reconfiguration Controller	PHY input port

Related Links

Running the Auto Sweep Test on page 297

12.3.2.5 PMA Analog Setting Control Configuration (Stratix V)

Use the following configuration to control PMA Analog settings in Stratix V designs.

Figure 145. System Configuration: PMA Analog Setting Control (Stratix V)

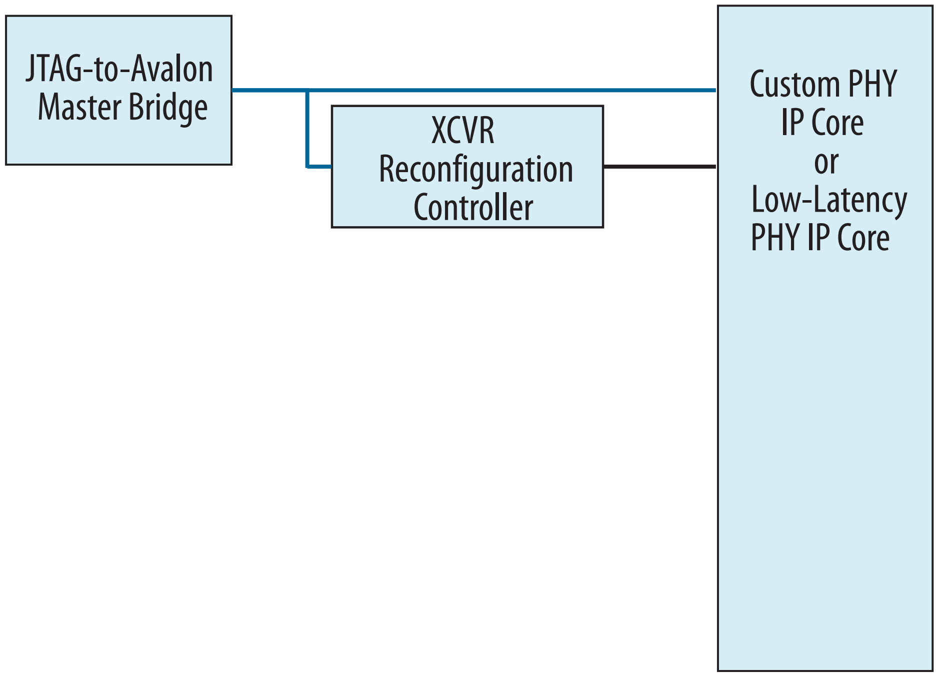

JTAG-to-Avalon Master Bridge
XCVR Reconfiguration Controller
Custom PHY IP Core or Low-Latency PHY IP Core

Table 88. System Connections: PMA Analog Setting Control (Stratix V)

From	To
JTAG to Avalon Master Bridge	Transceiver Reconfiguration Controller
JTAG to Avalon Master Bridge	PHY input port
Transceiver Reconfiguration Controller	PHY input port

Related Links

Controlling PMA Analog Settings on page 297

12.3.3 Instantiating and Parameterizing Intel Arria 10 Debug IP cores

To debug Intel Arria 10 designs with the Transceiver Toolkit, you must enable debugging settings in Transceiver Intel FPGA IP cores. You can either activate these settings when you first instantiate these components, or modify your instance after preliminary compilation.

The IP cores that you modify are:

- Transceiver Native PHY
- Transceiver ATX PLL
- CMU PLL
- fPLL

The parameters that you enable in the debug IP cores are:

Table 89. IP Cores and Debug Settings

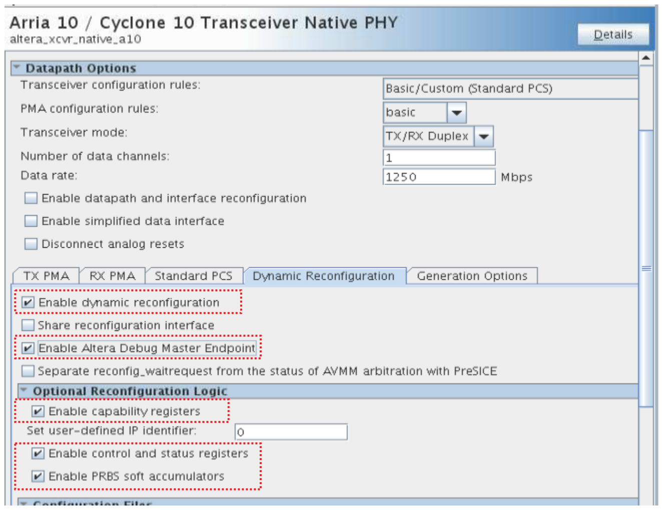
For more information about these parameters, refer to *Debug Settings for Transceiver IP Cores*.

IP Core	Enable dynamic reconfiguration	Enable Altera Debug Master Endpoint	Enable capability registers	Enable control and status registers	Enable PRBS Soft accumulators
Transceiver Native PHY	Yes	Yes	Yes	Yes	Yes
Transceiver ATX PLL	Yes	Yes			
CMU PLL	Yes	Yes			
fPLL	Yes	Yes			

For each transceiver IP core:

1. In the **IP Components** tab of the Project Navigator, right-click the IP instance, and click **Edit in Parameter Editor**.
2. Turn on debug settings as they appear in the *IP Cores and Debug Settings* table above.

Figure 146. Intel Arria 10 Transceiver Native PHY IP Core in the Parameter Editor



Arria 10 / Cyclone 10 Transceiver Native PHY
altera_xcvr_native_a10

Datapath Options

Transceiver configuration rules: Basic/Custom (Standard PCS)
PMA configuration rules: basic
Transceiver mode: TX/RX Duplex
Number of data channels: 1
Data rate: 1250 Mbps

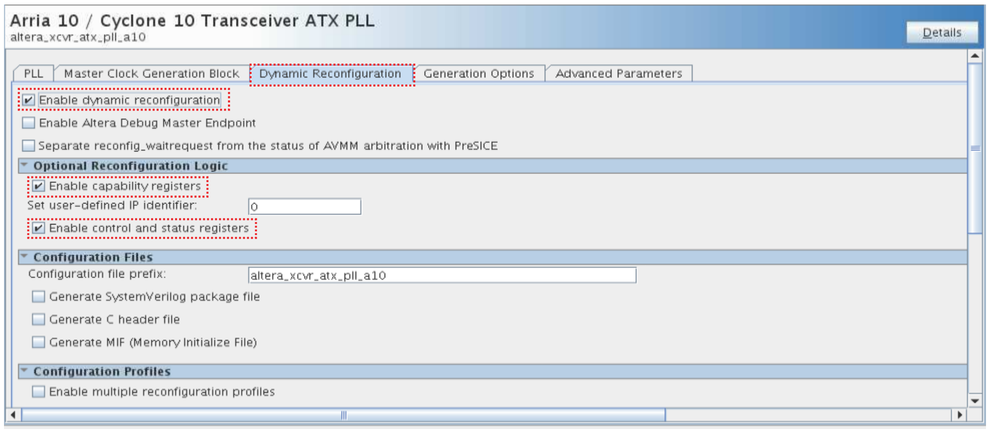
☐ Enable datapath and interface reconfiguration
☐ Enable simplified data interface
☐ Disconnect analog resets

☑ Enable dynamic reconfiguration
☐ Share reconfiguration interface
☑ Enable Altera Debug Master Endpoint
☐ Separate reconfig_waitrequest from the status of AVMM arbitration with PreSICE

Optional Reconfiguration Logic

☑ Enable capability registers
Set user-defined IP identifier: 0
☑ Enable control and status registers
☑ Enable PRBS soft accumulators

Figure 147. Intel Arria 10 Transceiver ATX PLL Core in the Parameter Editor



Arria 10 / Cyclone 10 Transceiver ATX PLL
altera_xcvr_atx_pll_a10

- [✓] Enable dynamic reconfiguration
- [ ] Enable Altera Debug Master Endpoint
- [ ] Separate reconfig_waitrequest from the status of AVMM arbitration with PreSICE

Optional Reconfiguration Logic

- [✓] Enable capability registers
- Set user-defined IP identifier: 0
- [✓] Enable control and status registers

Configuration Files

- Configuration file prefix: altera_xcvr_atx_pll_a10
- [ ] Generate SystemVerilog package file
- [ ] Generate C header file
- [ ] Generate MIF (Memory Initialize File)

Configuration Profiles

- [ ] Enable multiple reconfiguration profiles

3. Click **Generate HDL**.

After enabling parameters for all IPs in the design, recompile your project.

12.3.3.1 Debug Settings for Transceiver IP Cores

The table describes the settings that you turn on when preparing your transceiver for debug:

Table 90. Intel FPGA IP Settings for Transceiver Debug

Setting	Description
Enable Dynamic Reconfiguration	Allows you to change the behavior of the transceiver channels and PLLs without powering down the device
Enable Altera Debug Master Endpoint	Allows you to access the transceiver and PLL registers through System Console. When you recompile your design, Intel Quartus Prime software inserts the ADME, debug fabric, and embedded logic during synthesis.
Enable capability registers	Capability registers provide high level information about the configuration of the transceiver channel
Enable control and status registers	Enables soft registers to read status signals and write control signals on the PHY interface through the embedded debug.
Enable PRBS Soft Accumulators	Enables soft logic for performing PRBS bit and error accumulation when you use the hard PRBS generator and checker.

For more information about dynamic reconfiguration parameters on Intel Arria 10 devices, refer to the *Intel Arria 10 Transceiver PHY User Guide*.

Related Links

Dynamic Reconfiguration Parameters
In *Intel Arria 10 Transceiver PHY User Guide*

12.4 Programming the Design into an Intel FPGA

After you include debug components in the design, compile, and generate programming files, you can program the design in the Intel FPGA.

Related Links

Programming Intel FPGA Devices on page 444

12.5 Loading the Design in the Transceiver Toolkit

If the FPGA is already programmed with the project when loading, the Transceiver Toolkit automatically links the design to the target hardware in the toolkit. The toolkit automatically discovers links between transmitter and receiver of the same channel.

Before loading the device, ensure that you connect the hardware. The device and JTAG connections appear in the **Device** and **Connections** folders of the **System Explorer** pane.

To load the design into the Transceiver Toolkit:

1. In the System Console, click **File ➤ Load Design**.
2. Select the `.sof` programming file for the transceiver design.

After loading the project, the **designs** and **design instances** folders in the **System Explorer** pane display information about the design, such as the design name and the blocks in the design that can communicate to the System Console.

Related Links

System Explorer Pane on page 198

12.6 Linking Hardware Resources

Linking the hardware resources maps the project you load to the target FPGA. If you load multiple design projects for multiple FPGAs, then linking indicates which of the projects is in each of the FPGAs. The toolkit automatically discovers hardware and designs you connect. You can also manually link a design to connected hardware resources in the **System Explorer**.

If you are using more than one Intel FPGA board, you can set up a test with multiple devices linked to the same design. This setup is useful if you want to perform a link test between a transmitter and receiver on two separate devices. You can also load multiple Intel Quartus Prime projects and link between different systems. You can perform tests on separate and unrelated systems in a single Intel Quartus Prime instance.

Figure 148. One Channel Loopback Mode for Stratix V (28nm)

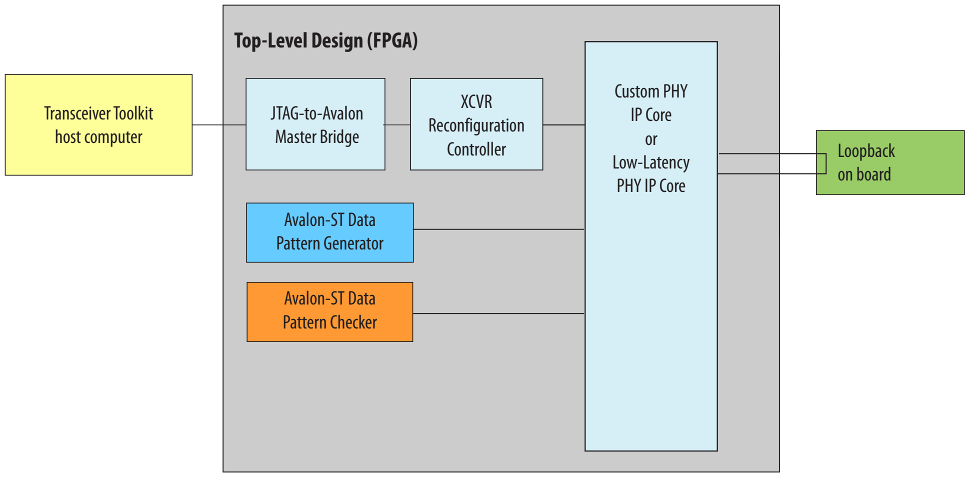

Top-Level Design (FPGA)
Transceiver Toolkit host computer
JTAG-to-Avalon Master Bridge
XCVR Reconfiguration Controller
Custom PHY IP Core or Low-Latency PHY IP Core
Loopback on board
Avalon-ST Data Pattern Generator
Avalon-ST Data Pattern Checker

Figure 149. One Channel Loopback Mode for Intel Arria 10devices

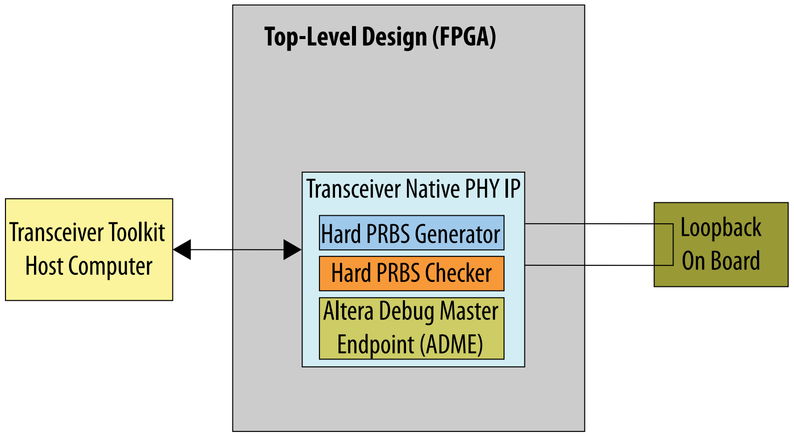

Top-Level Design (FPGA)
Transceiver Toolkit Host Computer
Transceiver Native PHY IP
Hard PRBS Generator
Hard PRBS Checker
Altera Debug Master Endpoint (ADME)
Loopback On Board

Figure 150. Four Channel Loopback Mode for Stratix V / 28nm

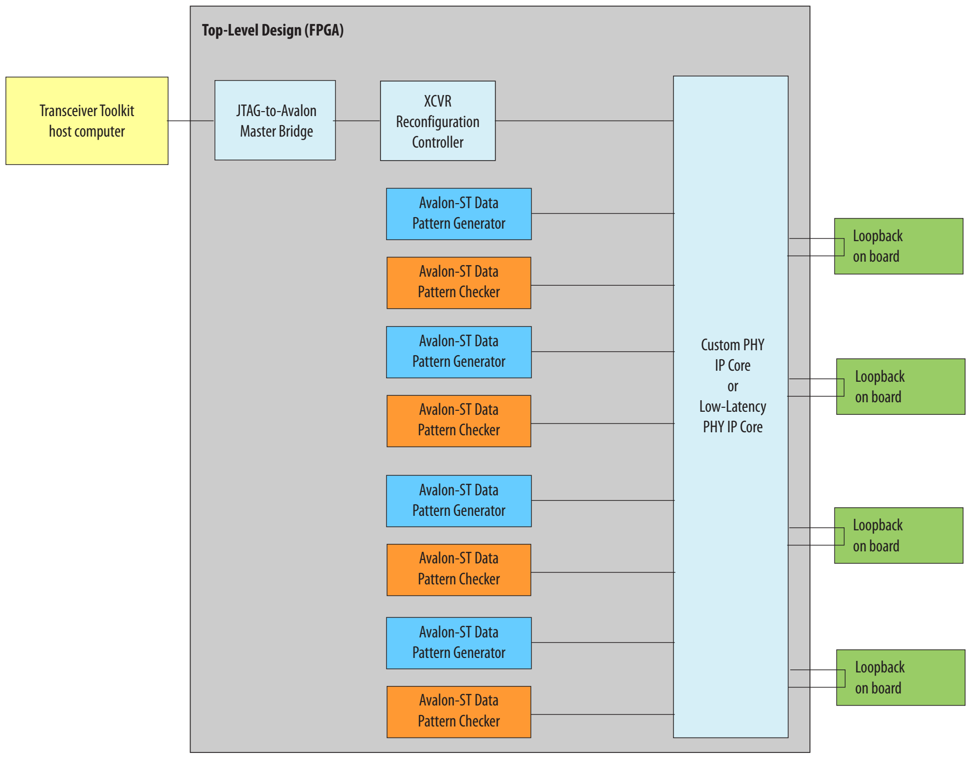
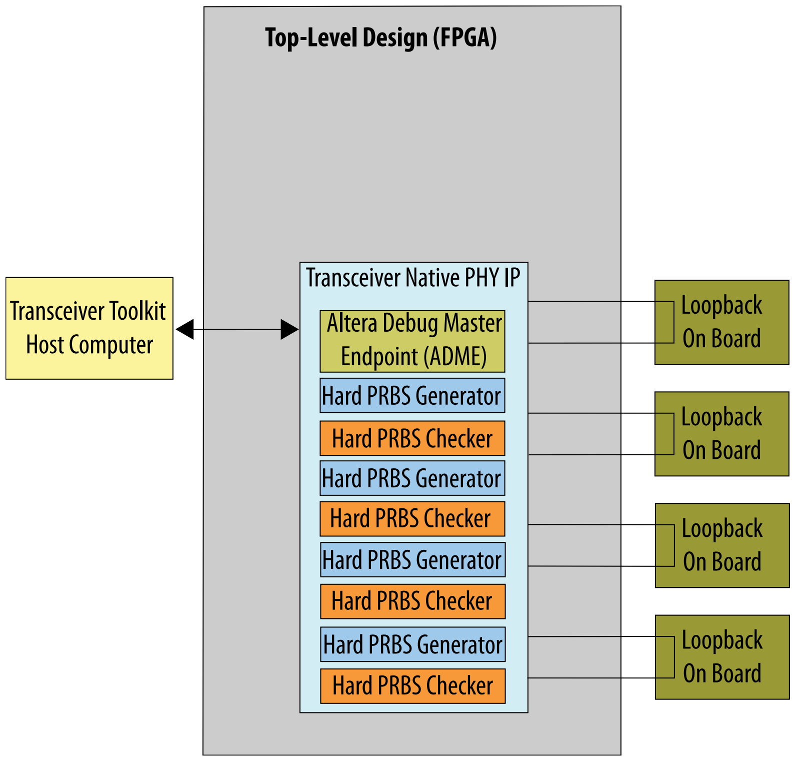

Top-Level Design (FPGA)
Transceiver Toolkit host computer
JTAG-to-Avalon Master Bridge
XCVR Reconfiguration Controller
Avalon-ST Data Pattern Generator
Avalon-ST Data Pattern Checker
Avalon-ST Data Pattern Generator
Avalon-ST Data Pattern Checker
Avalon-ST Data Pattern Generator
Avalon-ST Data Pattern Checker
Avalon-ST Data Pattern Generator
Avalon-ST Data Pattern Checker
Custom PHY IP Core or Low-Latency PHY IP Core
Loopback on board
Loopback on board
Loopback on board
Loopback on board

Figure 151. Four Channel Loopback Mode for Intel Arria 10devices



12.6.1 Linking One Design to One Device

To link one design to one device by one Intel FPGA Download Cable:

1. Load the design for your Intel Quartus Prime project.
2. If the design is not auto-linked, link each device to an appropriate design.
3. Create the link between channels on the device to test.

12.6.2 Linking Two Designs to Two Devices

To link two designs to two separate devices on the same board, connected by one Intel FPGA Download Cable download cable:

1. Load the design for all the Intel Quartus Prime project files you need.
2. If the design does not auto-link, link each device to an appropriate design
3. Open the project for the second device.
4. Link the second device on the JTAG chain to the second design (unless the design auto-links).
5. Create a link between the channels on the devices you want to test.

12.6.3 Linking One Design on Two Devices

To link the same design on two separate devices, follow these steps:

1. In the Transceiver Toolkit, open the `.sof` you are using on both devices.
2. Link the first device to this design instance.
3. Link the second device to the design.
4. Create a link between the channels on the devices you want to test.

12.6.4 Linking Designs and Devices on Separate Boards

To link two designs to two separate devices on separate boards that connect to separate Intel FPGA Download Cables:

1. Load the design for all the Intel Quartus Prime project files you need.
2. If the design does not auto-link, link each device to an appropriate design.
3. Create the link between channels on the device to test.
4. Link the device connected to the second Intel FPGA Download Cable to the second design.
5. Create a link between the channels on the devices you want to test.

12.7 Verifying Hardware Connections

Verifying hardware connections before you perform link tests saves time in the work flow. Use the toolkit to send data patterns and receive them correctly.

After you load the design and link the hardware:

1. Verify that the channels connect correctly and loop back properly on the hardware.
2. Verify that the RX is locked to Data.
3. Start the generator on the Transmitter Channel.
4. Start the checker on the Receiver Channel.
5. Verify you have Lock to Data, and the Bit Error Rate between the two is very low or zero.

After you verify communication between transmitter and receiver, you can create a link between the two transceivers and perform Auto Sweep and Eye Viewer tests with this pair.

Note: The Transceiver Toolkit can perform Eye Viewer tests only on Stratix V devices.

12.8 Identifying Transceiver Channels

Verify if the Transceiver Toolkit detects your channels correctly. The toolkit identifies a channel automatically whenever a receiver and transmitter share a transceiver channel.

The Transceiver Toolkit automatically displays transmitter and receiver channels that it recognizes on the **Transmitter Channels** and **Receiver Channels** tabs of the Channel Manager. You can also manually identify the transmitter and receiver in a transceiver channel, and create a link between the two for testing.

12.8.1 Controlling Transceiver Channels

To adjust or monitor transmitter or receiver settings while the channels are running:

- In the **Transmitter Channels** tab, click **Control Transmitter Channel**
- In the **Receiver Channels** tab, click **Control Receiver Channel**.
- In the **Transceiver Links** tab, click **Control Receiver Channel**.

For example, you can transmit a data pattern across the transceiver link, and then report the signal quality of the data you receive.

12.8.2 Creating Links

The toolkit automatically creates links when a receiver and transmitter share a transceiver channel. You can also manually create and delete links between transmitter and receiver channels. You create links in the **Setup** dialog box.

Setup Dialog Box

Click **Setup** from the Channel Manager to open the **Setup** dialog box.

Table 91. Setup Dialog Box Menu

Command Name	Description	Enabled
Edit Transmitter Alias	Starts the inline edit of the alias of the selected row.	Only enabled if one row is selected.
Edit Receiver Alias	Starts the inline edit of the alias of the selected row.	Only enabled if one row is selected.
Edit Transceiver Link Alias	Starts the inline edit of the alias of the selected row.	Only enabled if one row is selected.
Copy	Copies the text of the selected rows to the clipboard. The text copied depends on the column clicked on. The text copied to the clipboard is newline delimited.	Enabled if one or more rows are selected.

12.8.3 Manually Creating a Transceiver Link

Creating a link designates which Transmitter and Receiver channels connect physically.

To create a transceiver link with the Channel Manager:

1. Click **Setup**.
2. Select the generator and checker you want to control.
3. Select the transmitter and receiver pair you want to control.
4. Click **Create Transceiver Link**.
5. Click **Close**

The toolkit names the link automatically, but you can rename it using a link alias and give the link a shorter, more meaningful name.

12.9 Running Link Tests

Once you identify the transceiver channels for debugging, you can run link tests. Use the **Transceiver Links** tab to control link tests.

When you run link tests, channel color highlights indicate the test status:

Table 92. Channel Color Highlights

Color	Transmitter Channel	Receiver Channel
Red	Channel is closed or generator clock is not running.	Channel is closed or checker clock is not running.
Green	Generator is sending a pattern.	Checker is checking and data pattern is locked.
Neutral (same color as background)	Channel is open, generator clock is running, and generator is not sending a pattern.	Channel is open, checker clock is running, and checker is not checking.
Yellow	N/A	Checker is checking and data pattern is not locked.

12.9.1 Running BER Tests

BER tests help you assess signal integrity. Follow these steps to run BER tests across a transceiver link:

1. In the Channel Manager, click **Control Transceiver Link**.
2. Specify a PRBS **Test pattern**
3. If your device supports setting a **Checker mode**, set to **Data pattern checker**.
4. Try different values of **Reconfiguration**, **Generator**, or **Checker** settings, if available.
5. Click **Start** to run the pattern with your settings.
6. If your device supports error injection, you can click **Inject Error** to inject error bits.
7. You can also **Reset** the counter, or **Stop** the test.

   Note: Intel Arria 10 devices do not support **Inject Error** if you use the hard PRBS Pattern Generator and Checker in the system configuration.

Related Links

Bit Error Rate Test Configuration (Stratix V) on page 280

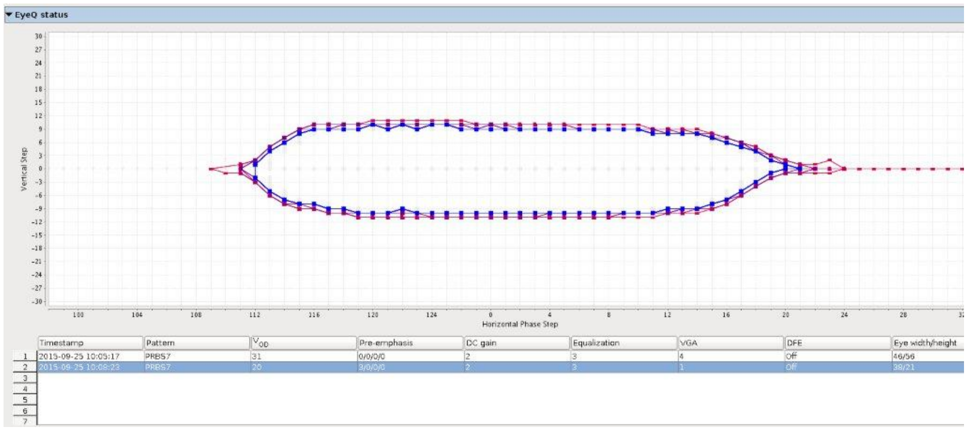
12.9.2 Signal Eye Margin Testing (Stratix V only)

Stratix V includes Eye Viewer circuitry, that allows visualization of the horizontal and vertical eye margin at the receiver. For supported devices, use signal eye tests to tune the PMA settings of your transceiver. This results in the best eye margin and BER at high data rates. The toolkit disables signal eye testing for unsupported devices.

The Eye Viewer graph can display a bathtub curve, eye diagram representing eye margin, or heat map display. The run list displays the statistics of each Eye Viewer test. When PMA settings are suitable, the bathtub curve is wide, with sharp slopes near the edges. The curve is up to 30 units wide. If the bathtub is narrow, then the signal quality is poor. The wider the bathtub curve, the wider the eye. The smaller the bathtub curve, the smaller the eye. The eye contour shows the estimated horizontal and vertical eye opening at the receiver.

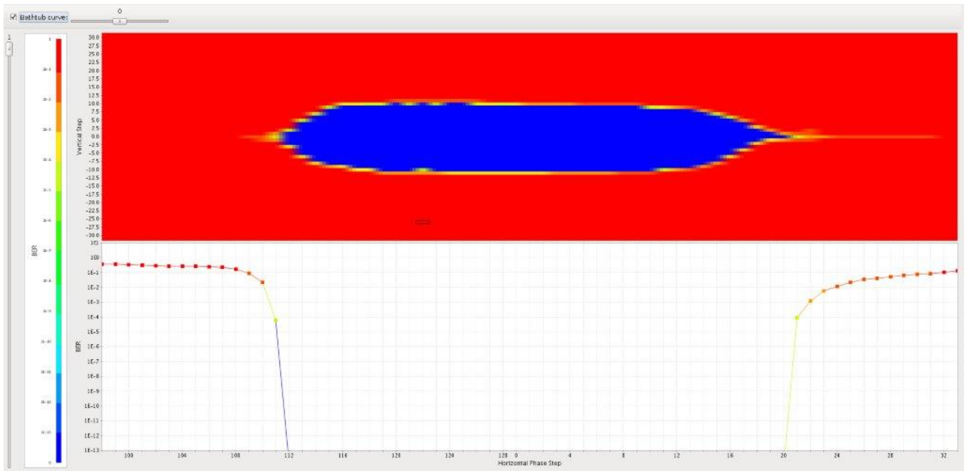
You can right-click any of the test runs in the list, and then click **Apply Settings to Device** to quickly apply that PMA setting to your device. You can also click **Export**, **Import**, or **Create Report**.

Figure 152. Eye Viewer Settings and Status Showing Results of Two Test Runs



	Timestamp	Pattern	V_{OD}	Pre-emphasis	DC gain	Equalization	VGA	DFE	Eye width/height
1	2015-09-25 10:05:17	PRBS7	31	0/0/0/0	2	3	4	Off	46/56
2	2015-09-25 10:08:23	PRBS7	20	3/0/0/0	2	3	1	Off	38/21
3									
4									
5									
6									
7									

Figure 153. Heat Map Display and Bathtub Curve Through Eye



12.9.2.1 Running PRBS Signal Eye Tests (Stratix V only)

Run PRBS signal eye tests to visualize the estimated horizontal and vertical eye opening at the receiver. After programming the FPGA with your debugging design, loading the design in the toolkit, and linking hardware, follow these steps to run PRBS signal eye tests:

1. Click **Setup**.
   a. Select the generator and checker you want to control.
   b. Select the transmitter and receiver pair you want to control.
   c. Click **Create Transceiver Link** and click **Close**.
2. Click **Link Eye Viewer**, and select **Eye Viewer** as the **Test mode**. The **Eye Viewer** mode displays test results as a bathtub curve, heat map, or eye contour representing bit error and phase offset data.
3. Specify the PRBS **Test pattern** and the **Checker mode**. Use **Serial bit comparator** checker mode only for checking a large window of error with custom traffic.

The checker mode option is only available after you turn on **Enable Eye Viewer block** and **Enable Bit Error Rate Block** in the Reconfiguration Controller component. (Stratix V designs only)

4. Specify **Run length** and **Eye Viewer settings** to control the test coverage and type of Eye Viewer results displayed, respectively.
5. Click **Start** to run the pattern with your settings. Eye Viewer uses the current channel settings to start a phase sweep of the channel. The phase sweep runs 32 iterations. As the run progresses, view the status under **Eye Viewer status**. Use this diagram to compare PMA settings for the same channel and to choose the best combination of PMA settings for a particular channel.
6. When the run completes, the chart displays the characteristics of each run. Click **Stop** to halt the test, change the PMA settings, and re-start the test. Click **Create Report** to export data to a table format for further viewing.

Related Links

- PRBS Signal Eye Test Configuration (Stratix V) on page 281
- AN 678: High-Speed Link Tuning Using Signal Conditioning Circuitry

12.9.3 Running Custom Traffic Tests (Stratix V only)

After programming the FPGA with your debugging design, loading the design in the toolkit, and linking hardware, follow these steps to run custom traffic tests:

1. In the Channel Manager, click **Setup**.
2. Select the associated reconfiguration controller.
3. Click **Create Transceiver Link** and click **Close**.
4. Click the **Receiver Eye Viewer** tab.
5. Select **Eye Viewer** as the **Test mode**. The **Eye Viewer** mode displays test results as a bathtub curve, heat map, or eye contour representing bit error and phase offset data.
6. Specify the PRBS **Test pattern**.
7. For **Checker mode**, select **Serial bit comparator**.

   The checker mode option is only available after you turn on **Enable Eye Viewer block** and **Enable Bit Error Rate Block** for the Reconfiguration Controller component.
8. Specify **Run length** and **Eye Viewer settings** to control the test coverage and type of Eye Viewer results displayed, respectively.
9. Click **Start** to run the pattern with your settings. Eye Viewer uses the current channel settings to start a phase sweep of the channel. The phase sweep runs 32 iterations. As the run progresses, view the status under **Eye Viewer status**.
10. When the run completes, the chart displays the characteristics of each run. Click **Stop** to halt the test, change the PMA settings, and re-start the test. Click **Create Report** to export data to a table format for further viewing.

Related Links

Custom Traffic Signal Eye Test Configuration (Stratix V) on page 282

12.9.4 Link Optimization Tests

The Transceiver Toolkit auto sweep test automatically sweeps PMA ranges to help you find the best transceiver settings. The toolkit allows you to store a history of the test runs, and keep a record of the best PMA settings. Use the best settings that the toolkit determined in your final design to improve signal integrity.

12.9.4.1 Running the Auto Sweep Test

to run link optimization tests:

1. In the **Transceiver Links** tab, select the channel you want to control.
2. Click **Link Auto Sweep**.
   The **Advanced** tab appears with **Auto sweep** as **Test mode**.
3. Specify the PRBS **Test pattern**.
4. Specify **Run length** experiment with the **Transmitter settings**, and **Receiver settings** to control the test coverage and PMA settings, respectively.
5. Click **Start** to run all combinations of tests meeting the PMA parameter limits. When the run completes the chart is displayed and the characteristics of each run are listed in the run list.
6. You can click **Stop** to halt the test, change the PMA settings, and re-start the test. Click **Create Report** to export data to a table format for further viewing.
7. If you want to determine the best tap settings using decision feedback equalization (DFE):
   a. Set the **DFE mode** to **Off**.
   b. Use Auto Sweep to find optimal PMA settings.
   c. If BER = 0, use the best PMA settings achieved.
   d. If BER > 0, use this PMA setting, and set the minimum and maximum values obtained from Auto Sweep to match this setting. Set the maximum DFE range to limits for each of the three DFE settings.
   e. Run **Create Report** to view the results and determine which DFE setting has the best BER. Use these settings in conjunction with the PMA settings for the best results.

Related Links

- Link Optimization Test Configuration (Stratix V) on page 283
- Instantiating and Parameterizing Intel Arria 10 Debug IP cores on page 285

12.9.4.2 TODO task

For the Altera Offline Compiler (AOC) to target a specific board package, you have to set the environment variable *AOCL_BOARD_PACKAGE ROOT* to point to the directory where you set up the board installation environment.

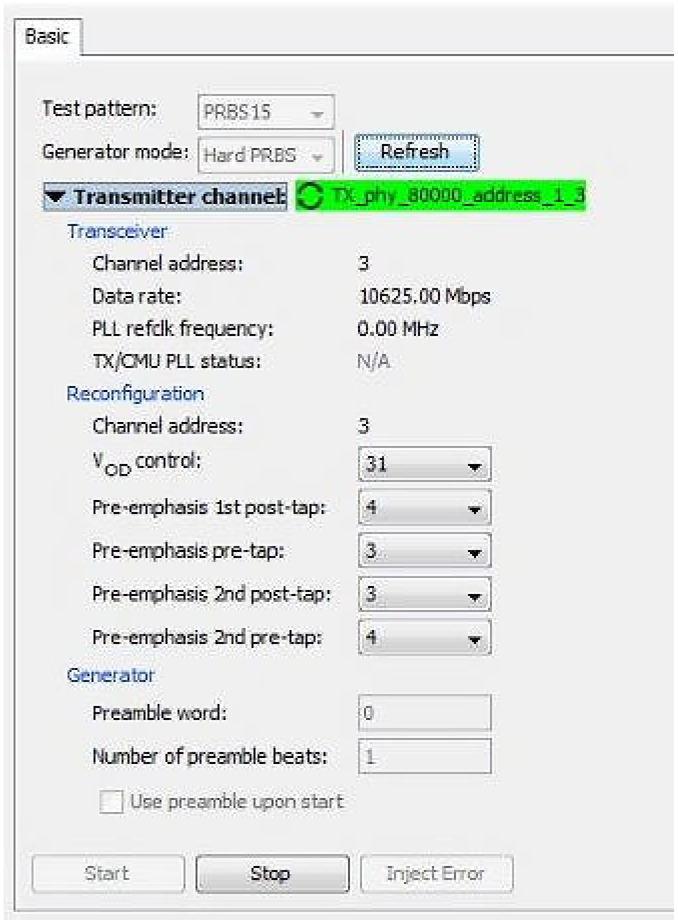
To set up the board environment, perform the following tasks:

12.10 Controlling PMA Analog Settings

The Transceiver Toolkit allows you to directly control PMA analog settings while the link is running. To control PMA analog settings, follow these steps:

1. In the Channel Manager, click **Setup**, and specify the following:
    a. In the **Transmitter Channels** tab, define a transmitter without a generator, and click **Create Transmitter Channel**.
    b. In the **Receiver Channels** tab, define a receiver without a generator, and click **Create Receiver Channel**.
    c. In the **Transceiver Links** tab, select the transmitter and receivers you want to control, and click **Create Transceiver Link**.
    d. Click **Close**.
2. Click **Control Receiver Channel**, **Control Transmitter Channel**, or **Control Transceiver Link** to directly control the PMA settings while running.

Figure 154. Controlling Transmitter Channel



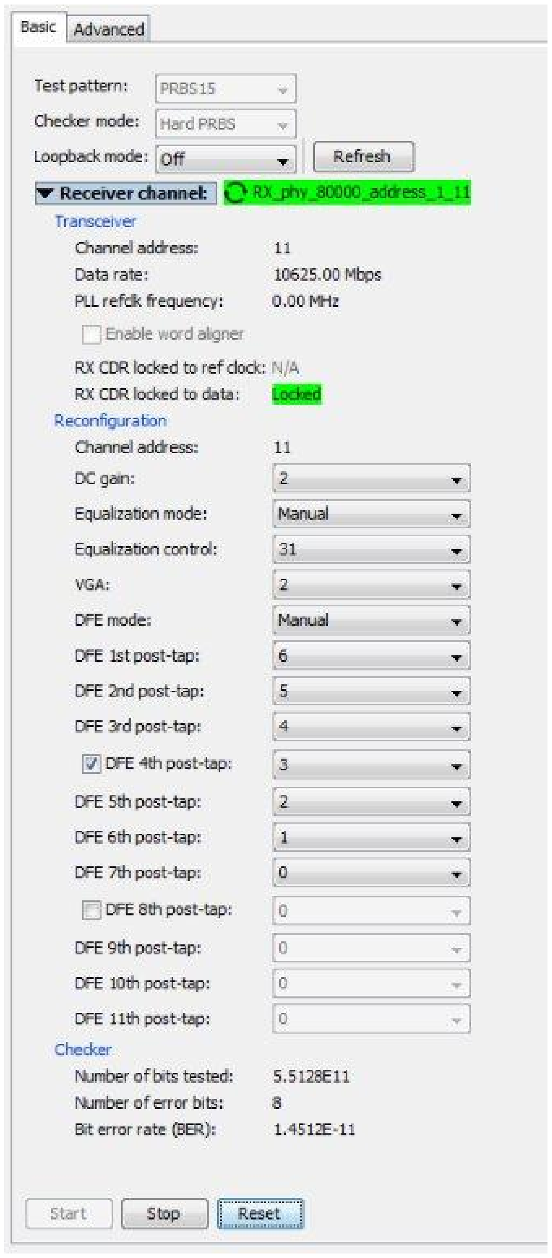
Test pattern: PRBS15
Generator mode: Hard PRBS
Transmitter channel: TX_phy_80000_address_1_3

Transceiver
Channel address: 3
Data rate: 10625.00 Mbps
PLL refclk frequency: 0.00 MHz
TX/CMU PLL status: N/A

Reconfiguration
Channel address: 3
V_{OD} control: 31
Pre-emphasis 1st post-tap: 4
Pre-emphasis pre-tap: 3
Pre-emphasis 2nd post-tap: 3
Pre-emphasis 2nd pre-tap: 4

Generator
Preamble word: 0
Number of preamble beats: 1
☐ Use preamble upon start

Figure 155. Controlling Receiver Channel



Test pattern: PRBS15
Checker mode: Hard PRBS
Loopback mode: Off

Receiver channel: RX_phy_80000_address_1_11

Transceiver

Channel address:	11
Data rate:	10625.00 Mbps
PLL refclk frequency:	0.00 MHz
☐ Enable word aligner	
RX CDR locked to ref clock:	N/A
RX CDR locked to data:	Locked

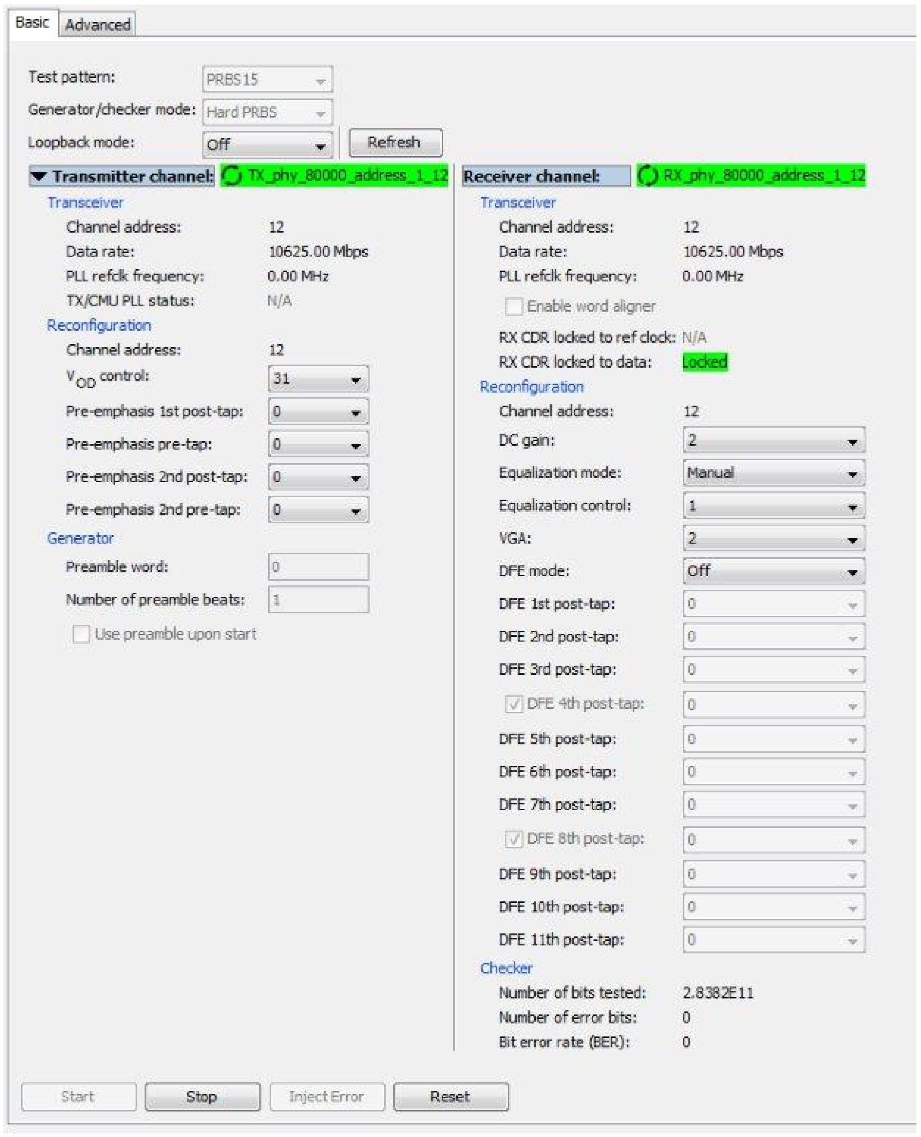
Reconfiguration

Channel address:	11
DC gain:	2
Equalization mode:	Manual
Equalization control:	31
VGA:	2
DFE mode:	Manual
DFE 1st post-tap:	6
DFE 2nd post-tap:	5
DFE 3rd post-tap:	4
☑ DFE 4th post-tap:	3
DFE 5th post-tap:	2
DFE 6th post-tap:	1
DFE 7th post-tap:	0
☐ DFE 8th post-tap:	0
DFE 9th post-tap:	0
DFE 10th post-tap:	0
DFE 11th post-tap:	0

Checker

Number of bits tested:	5.5128E11
Number of error bits:	8
Bit error rate (BER):	1.4512E-11

Figure 156. Controlling Transceiver Link



Test pattern: PRBS15
Generator/checker mode: Hard PRBS
Loopback mode: Off

Transmitter channel: TX_phy_80000_address_1_12

Transceiver
Channel address: 12
Data rate: 10625.00 Mbps
PLL refclk frequency: 0.00 MHz
TX/CMU PLL status: N/A

Reconfiguration
Channel address: 12
V_{OD} control: 31
Pre-emphasis 1st post-tap: 0
Pre-emphasis pre-tap: 0
Pre-emphasis 2nd post-tap: 0
Pre-emphasis 2nd pre-tap: 0

Generator
Preamble word: 0
Number of preamble beats: 1
☐ Use preamble upon start

Receiver channel: RX_phy_80000_address_1_12

Transceiver
Channel address: 12
Data rate: 10625.00 Mbps
PLL refclk frequency: 0.00 MHz
☐ Enable word aligner
RX CDR locked to ref clock: N/A
RX CDR locked to data: Locked

Reconfiguration
Channel address: 12
DC gain: 2
Equalization mode: Manual
Equalization control: 1
VGA: 2
DFE mode: Off
DFE 1st post-tap: 0
DFE 2nd post-tap: 0
DFE 3rd post-tap: 0
✓ DFE 4th post-tap: 0
DFE 5th post-tap: 0
DFE 6th post-tap: 0
DFE 7th post-tap: 0
✓ DFE 8th post-tap: 0
DFE 9th post-tap: 0
DFE 10th post-tap: 0
DFE 11th post-tap: 0

Checker
Number of bits tested: 2.8382E11
Number of error bits: 0
Bit error rate (BER): 0

Related Links

- Instantiating and Parameterizing Intel Arria 10 Debug IP cores on page 285
- PMA Analog Setting Control Configuration (Stratix V) on page 284
- User Interface Settings Reference on page 300

12.11 User Interface Settings Reference

The Transceiver Toolkit user interface allows you to specify these settings:

Note: All the settings appear in the **Transceiver Link** control pane.

Table 93. Transceiver Toolkit Control Pane Settings

Setting	Description	Control Pane
Alias	Name you choose for the channel.	Transmitter pane Receiver pane
Auto Sweep status	Reports the current and best tested bits, errors, bit error rate, and case count for the current Auto Sweep test.	Receiver pane
Bit error rate (BER)	Specifies errors divided by bits tested since the last reset of the checker.	Receiver pane
Channel address	Logical address number of the transceiver channel.	Transmitter pane Receiver pane
Data rate	Data rate of the channel that appears in the project file, or data rate the frequency detector measures. To use the frequency detector, turn on **Enable Frequency Counter** in the Data Pattern Checker IP core or Data Pattern Generator IP core, regenerate the IP cores, and recompile the design. The measured data rate depends on the Avalon management clock frequency that appears in the project file. If you make changes to your settings and want to sample the data rate again, click the **Refresh** button next to the **Data rate**	Transmitter pane Receiver pane
DC gain	Circuitry that provides an equal boost to the incoming signal across the frequency spectrum.	Receiver pane
DFE mode	Decision feedback equalization (DFE) for improving signal quality. • Values 1-5 (Stratix V devices) • Values 1-11 (Intel Arria 10 devices) In Intel Arria 10 devices, DFE modes are **Off**, **Manual** and **Adaptation Enabled**. DFE in **Adaptation Enabled** mode automatically tries to find the best tap values. In Stratix V devices DFE modes are **Off**, **Manual**, **One-time adaptive mode** and **Adaptation Enabled**. Adaptation Enabled mode DFE automatically tries to find the best tap values.	Receiver pane
Enable word aligner (Stratix V only)	Forces the transceiver channel to align to the word you specify.	Receiver pane
Equalization control	Boosts the high-frequency gain of the incoming signal, thereby compensating for the low-pass filter effects of the physical medium. When you use this option with DFE, use DFE in **Manual** or **Adaptation Enabled** mode. In Stratix V devices, auto sweep supports AEQ one-time adaptation.	Receiver pane
Equalization mode	For Intel Arria 10 devices, you can set **Equalization Mode** to **Manual** or **Triggered**. In Stratix V devices, Adaptive equalization (AEQ) automatically evaluates and selects the best combination of equalizer settings and turns off **Equalization Control**. The one-time selection determines the best setting and stops searching. You can use AEQ for multiple, independently controlled receiver channels.	Receiver pane
Error rate limit	Turns on or off error rate limits. **Start checking after** specifies the number of bits the toolkit waits before looking at the bit error rate (BER) for the next two checks. **Bit error rate achieves below** sets upper bit error rate limits. If the error rate is better than the set error rate, the test ends. **Bit error rate exceeds** sets lower bit error rate limits. If the error rate is worse than the set error rate, the test ends.	Receiver pane
Generator/Checker mode	Specifies **Data pattern checker** or **Serial bit comparator** for BER tests.	Receiver pane

continued...

Setting	Description	Control Pane
	If you enable **Serial bit comparator** the Data Pattern Generator sends the PRBS pattern, but the serial bit comparator checks the pattern. In **Bypass mode**, clicking **Start** begins counting on the Serial bit comparator. For BER testing: • Intel Arria 10 devices support the Data Pattern Checker and the Hard PRBS. • Stratix V devices support the Data Pattern Checker and the Serial Bit Checker.	
Horizontal phase step interval (Stratix V only)	Specifies the number of horizontal steps to increment when performing a sweep. Increasing the value increases the speed of the test but at a lower resolution. This option only applies to eye contour.	Transmitter pane Receiver pane
Increase test range	For the selected set of controls, increases the span of tests by one unit down for the minimum, and one unit up for the maximum. You can span either PMA Analog controls (non-DFE controls), or the DFE controls. You can quickly set up a test to check if any PMA setting combinations near your current best yields better results. To use, right-click the **Advanced** panel	Receiver pane
Inject Error (Stratix V only feature)	Flips one bit to the output of the data pattern generator to introduce an artificial error.	Transmitter pane
Maximum tested bits	Sets the maximum number of bits tested for each test iteration.	Receiver pane
Number of bits tested	Specifies the number of bits tested since the last reset of the checker.	Receiver pane
Number of error bits	Specifies the number of error bits encountered since the last reset of the checker.	Receiver pane
Number of preamble beats	(Stratix V only feature) Number of clock cycles to which the preamble word is sent before the test pattern begins.	Transmitter pane
PLL refclk freq	Channel reference clock frequency that appears in the project file, or reference clock frequency calculated from the measured data rate.	Transmitter pane Receiver pane
Populate with	Right-click the **Advanced** panel to load current values on the device as a starting point, or initially load the best settings auto sweep determines. The Intel Quartus Prime software automatically applies the values you specify in the drop-down lists for the Transmitter settings and Receiver settings.	Receiver pane
Preamble word	Word to send out if you use the preamble mode (only if you use soft PRBS Data Pattern Generator and Checker).	Transmitter pane
Pre-emphasis	This programmable module boosts high frequencies in the transmit data for each transmit buffer signal. This action counteracts possible attenuation in the transmission media. (Stratix V only) Using pre-emphasis can maximize the data eye opening at the far-end receiver.	Transmitter pane
Receiver channel	Specifies the name of the selected receiver channel.	Receiver pane
Refresh Button	After loading the `.pof`, loads fresh settings from the registers after running dynamic reconfiguration.	Transmitter pane Receiver pane
Reset	Resets the current test.	Receiver pane
Rules Based Configuration (RBC) validity checking	Displays in red any invalid combination of settings for each list under **Transmitter settings** and **Receiver settings**, based on previous settings.	Receiver pane

continued...

Setting	Description	Control Pane
	When you enable this option, the settings appear in red to indicate the current combination is invalid. This action avoids manually testing invalid settings that you cannot compile for your design, and prevents setting the device into an invalid mode for extended periods of time and potentially damaging the circuits.	
Run length	Sets coverage parameters for test runs.	Transmitter pane Receiver pane
RX CDR PLL status[10]	Shows the receiver in lock-to-reference (LTR) mode. When in auto-mode, if data cannot be locked, this signal alternates in LTD mode if the CDR is locked to data.	Receiver pane
RX CDR data status	Shows the receiver in lock-to-data (LTD) mode. When in auto-mode, if data cannot be locked, the signal stays high when locked to data and never switches.	Receiver pane
Serial loopback enabled	Inserts a serial loopback before the buffers, allowing you to form a link on a transmitter and receiver pair on the same physical channel of the device.	Transmitter pane Receiver pane
Start	Starts the pattern generator or checker on the channel to verify incoming data.	Transmitter pane Receiver pane
Stop	Stops generating patterns and testing the channel.	Transmitter pane Receiver pane
Target bit error rate (Stratix V only)	Finds the contour edge of the bit error rate that you select. This option only applies to eye contour mode.	Transmitter pane Receiver pane
Test mode	Allows you to specify the test mode. Intel Arria 10 devices support **Auto Sweep** test mode only. Stratix V devices support: • **Auto Sweep** test mode • **Eye Viewer** test mode • **Auto Sweep and Eye Viewer** test mode	Receiver pane
Test pattern	Test pattern sent by the transmitter channel. Intel Arria 10 devices support **PRBS9**, **PRBS15**, **PRBS23**, and **PRBS31**). Stratix V devices support **PRBS7**, **PRBS15**, **PRBS23**, **PRBS31**, **LowFrequency**, **HighFrequency**, and **Bypass mode**. The Data Pattern Checker self-aligns both high and low frequency patterns. Use **Bypass mode** to send user-design data.	Transmitter pane Receiver pane
Time limit	Specifies the time limit unit and value to have a maximum bounds time limit for each test iteration	Receiver
Transmitter channel	Specifies the name of the selected transmitter channel.	Transmitter pane
TX/CMU PLL status	Provides status of whether the transmitter channel PLL is locked to the reference clock.	Transmitter pane
Use preamble upon start	If turned on, sends the preamble word before the test pattern. If turned off, starts sending the test pattern immediately.	Transmitter pane
Vertical phase step interval (Stratix V only)	Specify the number of vertical steps to increment when performing a sweep. Increasing the value increases the speed of the test but at a lower resolution. This option only applies to the eye contour.	Transmitter pane Receiver pane
V_{OD} control	Programmable transmitter differential output voltage.	Transmitter pane

[10] For Stratix V devices, the Phase Frequency Detector (PFD) is inactive in LTD mode. The `rx_is_lockedtoref` status signal turns on and off randomly, and is not significant in LTD mode.

Related Links

Channel Manager on page 274

12.12 Troubleshooting Common Errors

Missing high-speed link pin connections

- Check if the pin connections to identify high-speed links (tx_p/n and rx_p/n) are missing. When porting an older design to the latest version of the Intel Quartus Prime software, make sure that these connections exist after porting.

Reset Issues:

- Ensure that the reset input to the Transceiver Native PHY, Transceiver Reset Controller, and ATX PLL IP cores is not held active (`1'b1`). The Transceiver Toolkit highlights in red all the Transceiver Native PHY channels that you are setting up.

Unconnected `reconfig_clk`

- You must connect and drive the `reconfig_clk` input to the Transceiver Native PHY and ATX PLL IP cores. Otherwise, the toolkit does not display the transceiver link channel.

12.13 Scripting API Reference

The Intel Quartus Prime software provides an API to access Transceiver Toolkit functions using Tcl commands, and script tasks such as linking device resources and identifying high-speed serial links.

To save your project setup in a Tcl script for use in subsequent testing sessions:

1. Set up and define links that describe the entire physical system
2. Click **Save Tcl Script** to save the setup for future use.

You can also build a custom test routine script.

To run the scripts, double-click the script name in the System Explorer scripts folder.

To view a list of the available Tcl command descriptions from the Tcl Console window, including example usage:

1. In the Tcl console, type `help help`. The Console displays all Transceiver Toolkit Tcl commands.
2. Type `help` <*command name*>. The Console displays the command description.

12.13.1 Transceiver Toolkit Commands

The following tables list the available Transceiver Toolkit scripting commands.

Table 94. Transceiver Toolkit channel_rx Commands

Command	Arguments	Function
`transceiver_channel_rx_get_data`	*<service-path>*	Returns a list of the current checker data. The results are in the order of number of bits, number of errors, and bit error rate.
`transceiver_channel_rx_get_dcgain`	*<service-path>*	Gets the DC gain value on the receiver channel.
`transceiver_channel_rx_get_dfe_tap_value`	*<service-path> <tap position>*	Gets the current tap value of the channel you specify at the tap position you specify.
`transceiver_channel_rx_get_eqctrl`	*<service-path>*	Gets the equalization control value on the receiver channel.
`transceiver_channel_rx_get_pattern`	*<service-path>*	Returns the current data checker pattern by name.
`transceiver_channel_rx_has_dfe`	*<service-path>*	Reports whether the channel you specify has the DFE feature available.
`transceiver_channel_rx_has_eye_viewer`	*<service-path>*	(Stratix V only) Reports whether the Eye Viewer feature is available for the channel you specify.
`transceiver_channel_rx_is_checking`	*<service-path>*	Returns non-zero if the checker is running.
`transceiver_channel_rx_is_dfe_enabled`	*<service-path>*	Reports whether the DFE feature is enabled on the channel you specify.
`transceiver_channel_rx_is_locked`	*<service-path>*	Returns non-zero if the checker is locked onto the incoming data.
`transceiver_channel_rx_reset_counters`	*<service-path>*	Resets the bit and error counters inside the checker.
`transceiver_channel_rx_reset`	*<service-path>*	Resets the channel you specify.
`transceiver_channel_rx_set_dcgain`	*<service-path> <value>*	Sets the DC gain value on the receiver channel.
`transceiver_channel_rx_set_dfe_enabled`	*<service-path> <disable(0)/ enable(1)>*	Enables or disables the DFE feature on the channel you specify.
`transceiver_channel_rx_set_dfe_tap_value`	*<service-path> <tap position> <tap value>*	Sets the current tap value of the channel you specify at the tap position you specify to the value you specify.
`transceiver_channel_rx_set_dfe_adaptive`	*<service-path> <adaptive-mode>*	Sets DFE adaptation mode of the channel you specify. **Value** / **Description** 0 / off 1 / adaptive 2 / one-time adaptive

continued...

Command	Arguments	Function
`transceiver_channel_rx_set_eqctrl`	*<service-path> <value>*	Sets the equalization control value on the receiver channel.
`transceiver_channel_rx_start_checking`	*<service-path>*	Starts the checker.
`transceiver_channel_rx_stop_checking`	*<service-path>*	Stops the checker.
`transceiver_channel_rx_get_eye_viewer_phase_step`	*<service-path>*	(Stratix V only) Gets the current phase step of the channel you specify.
`transceiver_channel_rx_set_pattern`	*<service-path> <pattern-name>*	Sets the expected pattern to the one specified by the pattern name.
`transceiver_channel_rx_is_eye_viewer_enabled`	*<service-path>*	(Stratix V only) Reports whether the Eye Viewer feature is enabled on the channel you specify.
`transceiver_channel_rx_set_eye_viewer_enabled`	*<service-path> <disable(0)/ enable(1)>*	(Stratix V only) Enables or disables the Eye Viewer feature on the channel you specify.
`transceiver_channel_rx_set_eye_viewer_phase_step`	*<service-path> <phase step>*	(Stratix V only) Sets the phase step of the channel you specify.
`transceiver_channel_rx_set_word_aligner_enabled`	*<service-path> <disable(0)/ enable(1)>*	Enables or disables the word aligner of the channel you specify.
`transceiver_channel_rx_is_word_aligner_enabled`	*<service-path> <disable(0)/ enable(1)>*	Reports whether the word aligner feature is enabled on the channel you specify.
`transceiver_channel_rx_is_locked`	*<service-path>*	Returns non-zero if the checker is locked onto the incoming signal.
`transceiver_channel_rx_is_rx_locked_to_data`	*<service-path>*	Returns 1 if transceiver is in lock to data (LTD) mode. Otherwise `0`.
`transceiver_channel_rx_is_rx_locked_to_ref`	*<service-path>*	Returns 1 if transceiver is in lock to reference (LTR) mode. Otherwise `0`.
`transceiver_channel_rx_has_eye_viewer_1d`	*<service-path>*	(Stratix V only) Detects whether the eye viewer in *<service-path>* supports 1D-Eye Viewer mode.
`transceiver_channel_rx_set_1deye_mode`	*<service-path> <disable(0)/ enable(1)>*	(Stratix V only) Enables or disables 1D-Eye Viewer mode.
`transceiver_channel_rx_get_1deye_mode`	*<service-path>*	(Stratix V only) Returns whether 1D-Eye Viewer mode is on or off.

Table 95. Transceiver Toolkit channel_tx Commands

Command	Arguments	Function
`transceiver_channel_tx_disable_preamble`	*<service-path>*	Disables the preamble mode at the beginning of generation.
`transceiver_channel_tx_enable_preamble`	*<service-path>*	Enables the preamble mode at the beginning of generation.
`transceiver_channel_tx_get_number_of_preamble_beats`	*<service-path>*	Returns the number of beats to send out the preamble word.
`transceiver_channel_tx_get_pattern`	*<service-path>*	Returns the pattern.

continued...

Command	Arguments	Function
`transceiver_channel_tx_get_preamble_word`	*<service-path>*	Returns the preamble word.
`transceiver_channel_tx_get_preemph0t`	*<service-path>*	Gets the pre-emphasis first pre-tap value on the transmitter channel.
`transceiver_channel_tx_get_preemph1t`	*<service-path>*	Gets the pre-emphasis first post-tap value on the transmitter channel.
`transceiver_channel_tx_get_preemph2t`	*<service-path>*	Gets the pre-emphasis second post-tap value on the transmitter channel.
`transceiver_channel_tx_get_preemph3t`	*<service-path>*	Gets the pre-emphasis second pre-tap value on the transmitter channel.
`transceiver_channel_tx_get_vodctrl`	*<service-path>*	Gets the V_{OD} control value on the transmitter channel.
`transceiver_channel_tx_inject_error`	*<service-path>*	Injects a 1-bit error into the generator's output.
`transceiver_channel_tx_is_generating`	*<service-path>*	Returns non-zero if the generator is running.
`transceiver_channel_tx_is_preamble_enabled`	*<service-path>*	Returns non-zero if preamble mode is enabled.
`transceiver_channel_tx_set_number_of_preamble_beats`	*<service-path> <number-of-preamble-beats>*	Sets the number of beats to send out the preamble word.
`transceiver_channel_tx_set_pattern`	*<service-path> <pattern-name>*	Sets the output pattern to the one specified by the pattern name.
`transceiver_channel_tx_set_preamble_word`	*<service-path> <preamble-word>*	Sets the preamble word to be sent out.
`transceiver_channel_tx_set_preemph0t`	*<service-path> <value>*	Sets the pre-emphasis first pre-tap value on the transmitter channel.
`transceiver_channel_tx_set_preemph1t`	*<service-path> <value>*	Sets the pre-emphasis first post-tap value on the transmitter channel.
`transceiver_channel_tx_set_preemph2t`	*<service-path> <value>*	Sets the pre-emphasis second post-tap value on the transmitter channel.
`transceiver_channel_tx_set_preemph3t`	*<service-path> <value>*	Sets the pre-emphasis second pre-tap value on the transmitter channel.
`transceiver_channel_tx_set_vodctrl`	*<service-path> <vodctrl value>*	Sets the V_{OD} control value on the transmitter channel.
`transceiver_channel_tx_start_generation`	*<service-path>*	Starts the generator.
`transceiver_channel_tx_stop_generation`	*<service-path>*	Stops the generator.

Table 96. Transceiver Toolkit Transceiver Toolkit debug_link Commands

Command	Arguments	Function
`transceiver_debug_link_get_pattern`	*<service-path>*	Gets the pattern the link uses to run the test.
`transceiver_debug_link_is_running`	*<service-path>*	Returns non-zero if the test is running on the link.
`transceiver_debug_link_set_pattern`	*<service-path> <data pattern>*	Sets the pattern the link uses to run the test.
`transceiver_debug_link_start_running`	*<service-path>*	Starts running a test with the currently selected test pattern.
`transceiver_debug_link_stop_running`	*<service-path>*	Stops running the test.

Table 97. Transceiver Toolkit reconfig_analog Commands

Command	Arguments	Function
`transceiver_reconfig_analog_get_logical_channel_address`	*<service-path>*	Gets the transceiver logic channel address currently set.
`transceiver_reconfig_analog_get_rx_dcgain`	*<service-path>*	Gets the DC gain value on the receiver channel specified by the current logic channel address.
`transceiver_reconfig_analog_get_rx_eqctrl`	*<service-path>*	Gets the equalization control value on the receiver channel specified by the current logic channel address.
`transceiver_reconfig_analog_get_tx_preemph0t`	*<service-path>*	Gets the pre-emphasis first pre-tap value on the transmitter channel specified by the current logic channel address.
`transceiver_reconfig_analog_get_tx_preemph1t`	*<service-path>*	Gets the pre-emphasis first post-tap value on the transmitter channel specified by the current logic channel address.
`transceiver_reconfig_analog_get_tx_preemph2t`	*<service-path>*	Gets the pre-emphasis second post-tap value on the transmitter channel specified by the current logic channel address.
`transceiver_reconfig_analog_get_tx_vodctrl`	*<service-path>*	Gets the V_{OD} control value on the transmitter channel specified by the current logic channel address.
`transceiver_reconfig_analog_set_logical_channel_address`	*<service-path> <logic channel address>*	Sets the transceiver logic channel address.
`transceiver_reconfig_analog_set_rx_dcgain`	*<service-path> <dc_gain value>*	Sets the DC gain value on the receiver channel specified by the current logic channel address
`transceiver_reconfig_analog_set_rx_eqctrl`	*<service-path> <eqctrl value>*	Sets the equalization control value on the receiver channel specified by the current logic channel address.
`transceiver_reconfig_analog_set_tx_preemph0t`	*<service-path> <value>*	Sets the pre-emphasis first pre-tap value on the transmitter channel specified by the current logic channel address.

continued...

Command	Arguments	Function
`transceiver_reconfig_analog_set_tx_preemph1t`	*<service-path> < value>*	Sets the pre-emphasis first post-tap value on the transmitter channel specified by the current logic channel address.
`transceiver_reconfig_analog_set_tx_preemph2t`	*<service-path> <value>*	Sets the pre-emphasis second post-tap value on the transmitter channel specified by the current logic channel address.
`transceiver_reconfig_analog_set_tx_vodctrl`	*<service-path> <vodctrl value>*	Sets the V_{OD} control value on the transmitter channel specified by the current logic channel address.

.

Table 98. Transceiver Toolkit Decision Feedback Equalization (DFE) Commands

Command	Arguments	Function
`alt_xcvr_reconfig_dfe_get_logical_channel_address`	*<service-path>*	Gets the logic channel address that other `alt_xcvr_reconfig_dfe` commands use to apply.
`alt_xcvr_reconfig_dfe_is_enabled`	*<service-path>*	Reports whether the DFE feature is enabled on the previously channel you specify.
`alt_xcvr_reconfig_dfe_set_enabled`	*<service-path> <disable(0)/ enable(1)>*	Enables or disables the DFE feature on the previously channel you specify.
`alt_xcvr_reconfig_dfe_set_logical_channel_address`	*<service-path> <logic channel address>*	(Stratix V only) Sets the logic channel address that other `alt_xcvr_reconfig_eye_viewer` commands use.
`alt_xcvr_reconfig_dfe_set_tap_value`	*<service-path> <tap position> <tap value>*	Sets the tap value at the previously channel you specify at specified tap position and value.

Table 99. Transceiver Toolkit Eye Monitor Commands (Stratix V only)

Command	Arguments	Function
`alt_xcvr_custom_is_word_aligner_enabled`	*<service-path> <disable(0)/ enable(1)>*	Reports whether the word aligner feature is enabled on the previously channel you specify.
`alt_xcvr_custom_set_word_aligner_enabled`	*<service-path> <disable(0)/ enable(1)>*	Enables or disables the word aligner of the previously channel you specify.
`alt_xcvr_custom_is_rx_locked_to_data`	*<service-path>*	Returns whether the receiver CDR is locked to data.
`alt_xcvr_custom_is_rx_locked_to_ref`	*<service-path>*	Returns whether the receiver CDR PLL is locked to the reference clock.
`alt_xcvr_custom_is_serial_loopback_enabled`	*<service-path>*	Returns whether the serial loopback mode of the previously channel you specify is enabled.

continued...

Command	**Arguments**	**Function**
`alt_xcvr_custom_set_serial_loopback_enabled`	*<service-path> <disable(0)/ enable(1)>*	Enables or disables the serial loopback mode of the previously channel you specify.
`alt_xcvr_custom_is_tx_pll_locked`	*<service-path>*	Returns whether the transmitter PLL is locked to the reference clock.
`alt_xcvr_reconfig_eye_viewer_get_logical_channel_address`	*<service-path>*	Gets the logic channel address on which other `alt_reconfig_eye_viewer` commands use.
`alt_xcvr_reconfig_eye_viewer_get_phase_step`	*<service-path>*	Gets the current phase step of the previously channel you specify.
`alt_xcvr_reconfig_eye_viewer_is_enabled`	*<service-path>*	Reports whether the Eye Viewer feature is enabled on the previously channel you specify.
`alt_xcvr_reconfig_eye_viewer_set_enabled`	*<service-path> <disable(0)/ enable(1)>*	Enables or disables the Eye Viewer feature on the previously channel you specify. Setting a value of 2 enables both Eye Viewer and the Serial Bit Comparator.
`alt_xcvr_reconfig_eye_viewer_set_logical_channel_address`	*<service-path> <logic channel address>*	Sets the logic channel address that other `alt_reconfig_eye_viewer` commands use.
`alt_xcvr_reconfig_eye_viewer_set_phase_step`	*<service-path> <phase step>*	Sets the phase step of the previously channel you specify.
`alt_xcvr_reconfig_eye_viewer_has_ber_checker`	*<service-path>*	Detects whether the eye viewer pointed to by *<service-path>* supports the Serial Bit Comparator.
`alt_xcvr_reconfig_eye_viewer_ber_checker_is_enabled`	*<service-path>*	Detects whether the Serial Bit Comparator is enabled.
`alt_xcvr_reconfig_eye_viewer_ber_checker_start`	*<service-path>*	Starts the Serial Bit Comparator counters.
`alt_xcvr_reconfig_eye_viewer_ber_checker_stop`	*<service-path>*	Stops the Serial Bit Comparator counters.
`alt_xcvr_reconfig_eye_viewer_ber_checker_reset_counters`	*<service-path>*	Resets the Serial Bit Comparator counters.
`alt_xcvr_reconfig_eye_viewer_ber_checker_is_running`	*<service-path>*	Reports whether the Serial Bit Comparator counters are currently running or not.
`alt_xcvr_reconfig_eye_viewer_ber_checker_get_data`	*<service-path>*	Gets the current total bit, error bit, and exception counts for the Serial Bit Comparator.

continued...

Command	Arguments	Function
`alt_xcvr_reconfig_eye_viewer_has_1deye`	*<service-path>*	Detects whether the eye viewer pointed to by *<service-path>* supports 1D-Eye Viewer mode.
`alt_xcvr_reconfig_eye_viewer_set_1deye_mode`	*<service-path> <disable(0)/ enable(1)*	Enables or disables 1D-Eye Viewer mode.
`alt_xcvr_reconfig_eye_viewer_get_1deye_mode`	*<service-path>*	Gets the enable or disabled state of 1D-Eye Viewer mode.

Table 100. Channel Type Commands

Command	Arguments	Function
`get_channel_type`	*<service-path> <logical-channel-num>*	Reports the detected type (GX/GT) of channel *<logical-channel-num>* for the reconfiguration block located at *<service-path>*.
`set_channel_type`	*<service-path> <logical-channel-num> <channel-type>*	Overrides the detected channel type of channel *<logical-channel-num>* for the reconfiguration block located at *<service-path>* to the type specified (0:GX, 1:GT).

Table 101. Loopback Commands

Command	Arguments	Function
`loopback_get`	*<service-path>*	Returns the value of a setting or result on the loopback channel. Available results include: • Status—running or stopped. • Bytes—number of bytes sent through the loopback channel. • Errors—number of errors reported by the loopback channel. • Seconds—number of seconds since the loopback channel was started.
`loopback_set`	*<service-path>*	Sets the value of a setting controlling the loopback channel. Some settings are only supported by particular channel types. Available settings include: • Timer—number of seconds for the test run. • Size—size of the test data. • Mode—mode of the test.
`loopback_start`	*<service-path>*	Starts sending data through the loopback channel.
`loopback_stop`	*<service-path>*	Stops sending data through the loopback channel.

12.13.2 Data Pattern Generator Commands

You can use Data Pattern Generator commands to control data patterns for debugging transceiver channels. You must instantiate the Data Pattern Generator component to support these commands.

Table 102. Soft Data Pattern Generator Commands

Command	Arguments	Function
`data_pattern_generator_start`	*<service-path>*	Starts the data pattern generator.
`data_pattern_generator_stop`	*<service-path>*	Stops the data pattern generator.
`data_pattern_generator_is_generating`	*<service-path>*	Returns non-zero if the generator is running.
`data_pattern_generator_inject_error`	*<service-path>*	Injects a 1-bit error into the generator output.

continued...

Command	Arguments	Function
`data_pattern_generator_set_pattern`	*<service-path> <pattern-name>*	Sets the output pattern specified by the *<pattern-name>*. In all, 6 patterns are available, 4 are pseudo-random binary sequences (PRBS), 1 is high frequency and 1 is low frequency. The PRBS7, PRBS15, PRBS23, PRBS31, HF (outputs high frequency, constant pattern of alternating 0s and 1s), and LF (outputs low frequency, constant pattern of 10b'1111100000 for 10-bit symbols and 8b'11110000 for 8-bit symbols) pattern names are defined. PRBS files are clear text and you can modify the PRBS files.
`data_pattern_generator_get_pattern`	*<service-path>*	Returns currently selected output pattern.
`data_pattern_generator_get_available_patterns`	*<service-path>*	Returns a list of available data patterns by name.
`data_pattern_generator_enable_preamble`	*<service-path>*	Enables the preamble mode at the beginning of generation.
`data_pattern_generator_disable_preamble`	*<service-path>*	Disables the preamble mode at the beginning of generation.
`data_pattern_generator_is_preamble_enabled`	*<service-path>*	Returns a non-zero value if preamble mode is enabled.
`data_pattern_generator_set_preamble_word`	*<preamble-word>*	Sets the preamble word (could be 32-bit or 40-bit).
`data_pattern_generator_get_preamble_word`	*<service-path>*	Gets the preamble word.
`data_pattern_generator_set_preamble_beats`	*<service-path><number-of-preamble- beats>*	Sets the number of beats to send out in the preamble word.
`data_pattern_generator_get_preamble_beats`	*<service-path>*	Returns the currently set number of beats to send out in the preamble word.
`data_pattern_generator_fcnter_start`	*<service-path><max-cycles>*	Sets the max cycle count and starts the frequency counter.
`data_pattern_generator_check_status`	*<service-path>*	Queries the data pattern generator for current status. Returns a bitmap indicating the status, with bits defined as follows: [0]-enabled, [1]-bypass enabled, [2]-avalon, [3]-sink ready, [4]-source valid, and [5]-frequency counter enabled.
`data_pattern_generator_fcnter_report`	*<service-path><force-stop>*	Reports the current measured clock ratio, stopping the counting first depending on *<force-stop>*.

Table 103. Hard Data Pattern Generator Commands

Command	Arguments	Function
`hard_prbs_generator_start`	*<service-path>*	Starts the specified generator.
`hard_prbs_generator_stop`	*<service-path>*	Stops the specified generator.

continued...

Command	Arguments	Function
`hard_prbs_generator_is_generating`	*<service-path>*	Checks the generation status. Returns 1 if generating, `0` otherwise.
`hard_prbs_generator_set_pattern`	*<service-path>* *<pattern>*	Sets the pattern of the specified hard PRBS generator to parameter `pattern`.
`hard_prbs_generator_get_pattern`	*<service-path>*	Returns the current pattern for a given hard PRBS generator.
`hard_prbs_generator_get_available_patterns`	*<service-path>*	Returns the available patterns for a given hard PRBS generator.

12.13.3 Data Pattern Checker Commands

You can use Data Pattern Checker commands to verify your generated data patterns. You must instantiate the Data Pattern Checker component to support these commands.

Table 104. Soft Data Pattern Checker Commands

Command	Arguments	Function
`data_pattern_checker_start`	*<service-path>*	Starts the data pattern checker.
`data_pattern_checker_stop`	*<service-path>*	Stops the data pattern checker.
`data_pattern_checker_is_checking`	*<service-path>*	Returns a non-zero value if the checker is running.
`data_pattern_checker_is_locked`	*<service-path>*	Returns non-zero if the checker is locked onto the incoming data.
`data_pattern_checker_set_pattern`	*<service-path>* *<pattern-name>*	Sets the expected pattern to the one specified by the *<pattern-name>*.
`data_pattern_checker_get_pattern`	*<service-path>*	Returns the currently selected expected pattern by name.
`data_pattern_checker_get_available_patterns`	*<service-path>*	Returns a list of available data patterns by name.
`data_pattern_checker_get_data`	*<service-path>*	Returns a list of the current checker data. The results are in the following order: number of bits, number of errors, and bit error rate.
`data_pattern_checker_reset_counters`	*<service-path>*	Resets the bit and error counters inside the checker.
`data_pattern_checker_fcnter_start`	*<service-path><max-cycles>*	Sets the max cycle count and starts the frequency counter.
`data_pattern_checker_check_status`	*<service-path>* *<service-path>*	Queries the data pattern checker for current status. Returns a bitmap indicating status, with bits defined as follows: [0]-enabled, [1]-locked, [2]-bypass enabled, [3]-avalon, [4]-sink ready, [5]-source valid, and [6]-frequency counter enabled.
`data_pattern_checker_fcnter_report`	*<service-path><force-stop>*	Reports the current measured clock ratio, stopping the counting first depending on *<force-stop>*.

Table 105. Hard Data Pattern Checker Commands

Command	Arguments	Function
`hard_prbs_checker_start`	*<service-path>*	Starts the specified hard PRBS checker.
`hard_prbs_checker_stop`	*<service-path>*	Stops the specified hard PRBS checker.
`hard_prbs_checker_is_checking`	*<service-path>*	Checks the running status of the specified hard PRBS checker. Returns a non-zero value if the checker is running.
`hard_prbs_checker_set_pattern`	*<service-path>* *<pattern>*	Sets the pattern of the specified hard PRBS checker to parameter *<pattern>*.
`hard_prbs_checker_get_pattern`	*<service-path>*	Returns the current pattern for a given hard PRBS checker.
`hard_prbs_checker_get_available_patterns`	*<service-path>*	Returns the available patterns for a given hard PRBS checker.
`hard_prbs_checker_get_data`	*<service-path>*	Returns the current bit and error count data from the specified hard PRBS checker.
`hard_prbs_checker_reset_counters`	*<service-path>*	Resets the bit and error counts of the specified hard PRBS checker.

12.14 Document Revision History

Table 106. Document Revision History

Date	Version	Changes
2017.11.06	17.1.0	• Renamed EyeQ to Eye Viewer. • Updated topic "Transceiver Debugging Flow" and renamed to "Transceiver Debugging Flow Walkthrough". • Updated instructions for instantiating and parameterizing Debug IP cores. — Removed figure: "Altera Debug Master Endpoint Block Diagram". • Added step on programming designs as a part of the debugging flow. • Updated information about debugging transceiver links.
2016.10.31	16.1.0	• Removed EyeQ support for Intel Arria 10. • Renamed *"Continuous Adaptation"* to *"Adaptation Enabled"*.
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*. • Added description of new Refresh button. • Added description of VGA dialog box. • Added two tables in Transceiver Toolkit Commands section. — Hard Data Pattern Generator Commands — Hard Data Pattern Checker Commands • Separated Arria 10 and Stratix V system configuration steps.

continued...

Date	Version	Changes
May 2015	15.0.0	• Added section about Implementation Differences Between Stratix V and Arria 10. • Added section about Recommended Flow for Arria 10 Transceiver Toolkit Design with the Intel Quartus Prime Software. • Added section about Transceiver Toolkit Troubleshooting • Updated the following sections with information about using the Transceiver Toolkit with Arria 10 devices: — Serial Bit Comparator Mode — Arria 10 Support and Limitations — Configuring BER Tests — Configuring PRBS Signal Eye Tests — Adapting Altera Design Examples — Modifying Design Examples — Configuring Custom Traffic Signal Eye Tests — Configuring Link Optimization Tests — Configuring PMA Analog Setting Control — Running BER Tests — Toolkit GUI Setting Reference • Reworked Table: Transceiver Toolkit IP Core Configuration • Replaced Figure: EyeQ Settings and Status Showing Results of Two Test Runs with Figure: EyeQ Settings and Status Showing Results of Three Test Runs. • Added Figure: Arria 10 Altera Debug Master Endpoint Block Diagram. • Added Figure: BER Test Configuration (Arria10/ Gen 10/ 20nm) Block Diagram. • Added Figure: PRBS Signal Test Configuration (Arria 10/ 20nm) Block Diagram. • Added Figure: Custom Traffic Signal Eye Test Configuration (Arria 10/ Gen 10/ 20nm) Block Diagram. • Added Figure: PMA Analog Setting Control Configuration (Arria 10/ Gen 10/ 20nm) Block Diagram. • Added Figure: One Channel Loopback Mode (Arria 10/ 20nm) Block Diagram. • Added Figure: Four Channel Loopback Mode (Arria 10/ Gen 10/ 20nm) Block Diagram. Software Version 15.0 Limitations • Transceiver Toolkit supports EyeQ for Arria 10 designs. • Supports optional hard acceleration for EyeQ. This allows for much faster EyeQ data collection. Enable this in the Arria 10 Transceiver Native PHY IP core under the **Dynamic Configuration** tab. Turn on **Enable ODI acceleration logic**.
December, 2014	14.1.0	• Added section about Arria 10 support and limitations.
June, 2014	14.0.0	• Updated GUI changes for Channel Manager with popup menus, IP Catalog, Intel Quartus Prime, and Qsys. • Added ADME and JTAG debug link info for Arria 10. • Added instructions to run Tcl script from command line. • Added heat map display option. • Added procedure to use internal PLL to generate reconfig_clk. • Added note stating RX CDR PLL status can toggle in LTD mode.
November, 2013	13.1.0	• Reorganization and conversion to DITA.
May, 2013	13.0.0	• Added Conduit Mode Support, Serial Bit Comparator, Required Files and Tcl command tables.
November, 2012	12.1.0	• Minor editorial updates. Added Tcl help information and removed Tcl command tables. Added 28-Gbps Transceiver support section.
August, 2012	12.0.1	• General reorganization and revised steps in modifying Altera example designs.
June, 2012	12.0.0	• Maintenance release for update of Transceiver Toolkit features.
November, 2011	11.1.0	• Maintenance release for update of Transceiver Toolkit features.
May, 2011	11.0.0	• Added new Tcl scenario.

continued...

Date	Version	Changes
December, 2010	10.1.0	• Changed to new document template. Added new 10.1 release features.
August, 2010	10.0.1	• Corrected links.
July 2010	10.0.0	• Initial release.

Related Links

Documentation Archive
For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

13 Quick Design Debugging Using Signal Probe

The Signal Probe incremental routing feature helps reduce the hardware verification process and time-to-market for system-on-a-programmable-chip (SOPC) designs. Easy access to internal device signals is important in the design or debugging process. The Signal Probe feature makes design verification more efficient by routing internal signals to I/O pins quickly without affecting the design. When you start with a fully routed design, you can select and route signals for debugging to either previously reserved or currently unused I/O pins.

The Signal Probe feature supports the Arria series, Cyclone series, MAX II, and Stratix series device families.

Related Links

System Debugging Tools Overview on page 183

13.1 Design Flow Using Signal Probe

The Signal Probe feature allows you to reserve available pins and route internal signals to those reserved pins, while preserving the behavior of your design. Signal Probe is an effective debugging tool that provides visibility into your FPGA.

You can reserve pins for Signal Probe and assign I/O standards after a full compilation. Each Signal Probe-source to Signal Probe-pin connection is implemented as an engineering change order (ECO) that is applied to your netlist after a full compilation.

To route the internal signals to the device’s reserved pins for Signal Probe, perform the following tasks:

1. Perform a full compilation.
2. Reserve Signal Probe Pins.
3. Assign Signal Probe sources.
4. Add registers between pipeline paths and Signal Probe pins.
5. Perform a Signal Probe compilation.
6. Analyze the results of a Signal Probe compilation.

13.1.1 Perform a Full Compilation

You must complete a full compilation to generate an internal netlist containing a list of internal nodes to probe.

To perform a full compilation, on the Processing menu, click **Start Compilation**.

13.1.2 Reserve Signal Probe Pins

Signal Probe pins can only be reserved after a full compilation. You can also probe any unused I/Os of the device. Assigning sources is a simple process after reserving Signal Probe pins. The sources for Signal Probe pins are the internal nodes and registers in the post-compilation netlist that you want to probe.

Note: Although you can reserve Signal Probe pins using many features within the Intel Quartus Prime software, including the Pin Planner and the Tcl interface, you should use the **Signal Probe Pins** dialog box to create and edit your Signal Probe pins.

13.1.3 Assign Signal Probe Sources

A Signal Probe source can be any combinational node, register, or pin in your post-compilation netlist. To find a Signal Probe source, in the Node Finder, use the Signal Probe filter to remove all sources that cannot be probed. You might not be able to find a particular internal node because the node can be optimized away during synthesis, or the node cannot be routed to the Signal Probe pin. For example, you cannot probe nodes and registers within Gigabit transceivers in Stratix IV devices because there are no physical routes available to the pins.

Note: To probe virtual I/O pins generated in low-level partitions in an incremental compilation flow, select the source of the logic that feeds the virtual pin as your Signal Probe source pin.

Because Signal Probe pins are implemented and routed as ECOs, turning the **Signal Probe enable** option on or off is the same as selecting **Apply Selected Change** or **Restore Selected Change** in the Change Manager window. If the Change Manager window is not visible at the bottom of your screen, on the View menu, point to **Utility Windows** and click **Change Manager**.

Related Links

- Engineering Change Management with the Chip Planner
  In *Intel Quartus Prime Standard Edition Handbook Volume 2*
- Signal Probe Pins Dialog Box
- Add Signal Probe Pins Dialog Box
  In Intel Quartus Prime Help

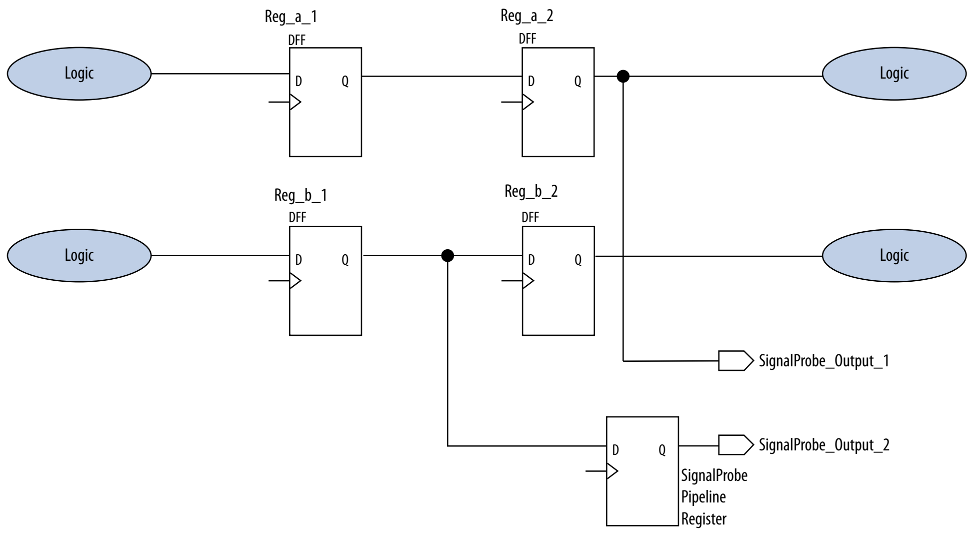
13.1.4 Add Registers Between Pipeline Paths and Signal Probe Pins

You can specify the number of registers placed between a Signal Probe source and a Signal Probe pin. The registers synchronize data to a clock and control the latency of the Signal Probe outputs. The Signal Probe feature automatically inserts the number of registers specified into the Signal Probe path.

The figure shows a single register between the Signal Probe source `Reg_b_1` and Signal Probe `Signal Probe_Output_2` output pin added to synchronize the data between the two Signal Probe output pins.

Note: When you add a register to a Signal Probe pin, the Signal Probe compilation attempts to place the register to best meet timing requirements. You can place Signal Probe registers either near the Signal Probe source to meet f_{MAX} requirements, or near the I/O to meet t_{CO} requirements.

Figure 157. Synchronizing Signal Probe Outputs with a Signal Probe Register



In addition to clock input for pipeline registers, you can also specify a reset signal pin for pipeline registers. To specify a reset pin for pipeline registers, use the Tcl command `make_sp`.

Related Links

Add Signal Probe Pins Dialog Box online help
    Information about how to pipeline an existing Signal Probe connection

13.1.5 Perform a Signal Probe Compilation

Perform a Signal Probe compilation to route your Signal Probe pins. A Signal Probe compilation saves and checks all netlist changes without recompiling the other parts of the design. A Signal Probe compilation takes a fraction of the time of a full compilation to finish. The design's current placement and routing are preserved.

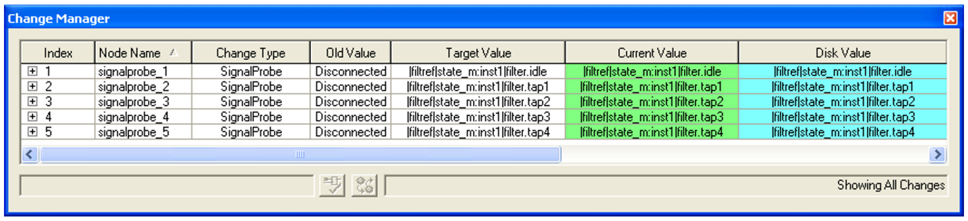
To perform a Signal Probe compilation, on the Processing menu, point to **Start** and click **Start Signal Probe Compilation**.

13.1.6 Analyze the Results of a Signal Probe Compilation

After a Signal Probe compilation, the results are available in the compilation report file. Each Signal Probe pin is displayed in the **Signal Probe Fitting Result** page in the **Fitter** section of the Compilation Report. To view the status of each Signal Probe pin in the **Signal Probe Pins** dialog box, on the Tools menu, click **Signal Probe Pins**.

The status of each Signal Probe pin appears in the Change Manager window . If the Change Manager window is not visible at the bottom of your GUI, from the View menu, point to **Utility Windows** and click **Change Manager**.

Figure 158. Change Manager Window with Signal Probe Pins



Change Manager

Index	Node Name	Change Type	Old Value	Target Value	Current Value	Disk Value
⊞ 1	signalprobe_1	SignalProbe	Disconnected	|filtref|state_m:inst1|filter.idle	|filtref|state_m:inst1|filter.idle	|filtref|state_m:inst1|filter.idle
⊞ 2	signalprobe_2	SignalProbe	Disconnected	|filtref|state_m:inst1|filter.tap1	|filtref|state_m:inst1|filter.tap1	|filtref|state_m:inst1|filter.tap1
⊞ 3	signalprobe_3	SignalProbe	Disconnected	|filtref|state_m:inst1|filter.tap2	|filtref|state_m:inst1|filter.tap2	|filtref|state_m:inst1|filter.tap2
⊞ 4	signalprobe_4	SignalProbe	Disconnected	|filtref|state_m:inst1|filter.tap3	|filtref|state_m:inst1|filter.tap3	|filtref|state_m:inst1|filter.tap3
⊞ 5	signalprobe_5	SignalProbe	Disconnected	|filtref|state_m:inst1|filter.tap4	|filtref|state_m:inst1|filter.tap4	|filtref|state_m:inst1|filter.tap4

Showing All Changes

To view the timing results of each successfully routed Signal Probe pin, on the Processing menu, point to **Start** and click **Start Timing Analysis**.

Related Links

Engineering Change Management with the Chip Planner documentation

13.1.7 What a Signal Probe Compilation Does

After a full compilation, you can start a Signal Probe compilation either manually or automatically. A Signal Probe compilation performs the following functions:

- Validates Signal Probe pins
- Validates your specified Signal Probe sources
- Adds registers into Signal Probe paths, if applicable
- Attempts to route from Signal Probe sources through registers to Signal Probe pins

To run the Signal Probe compilation immediately after a full compilation, on the Tools menu, click **Signal Probe Pins**. In the **Signal Probe Pins** dialog box, click **Start Check & Save All Netlist Changes**.

To run a Signal Probe compilation manually after a full compilation, on the Processing menu, point to **Start** and click **Start Signal Probe Compilation**.

Note: You must run the Fitter before a Signal Probe compilation. The Fitter generates a list of all internal nodes that can serve as Signal Probe sources.

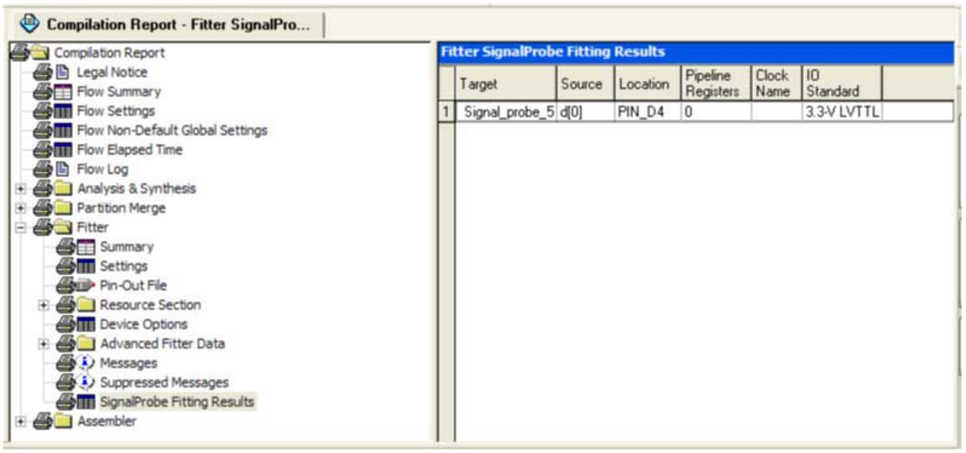
Turn the **Signal Probe enable** option on or off in the **Signal Probe Pins** dialog box to enable or disable each Signal Probe pin.

13.1.8 Understanding the Results of a Signal Probe Compilation

After a Signal Probe compilation, the results appear in two sections of the compilation report file. The fitting results and status of each Signal Probe pin appears in the **Signal Probe Fitting Result** screen in the Fitter section of the Compilation Report.

Table 107. Status Values

Status	Description
Routed	Connected and routed successfully
Not Routed	Not enabled
Failed to Route	Failed routing during last Signal Probe compilation
Need to Compile	Assignment changed since last Signal Probe compilation

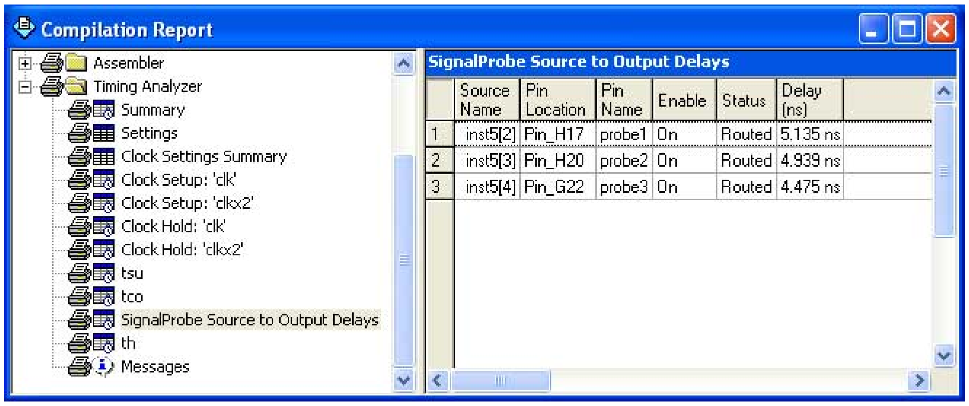


Fitter SignalProbe Fitting Results

	Target	Source	Location	Pipeline Registers	Clock Name	IO Standard
1	Signal_probe_5	d[0]	PIN_D4	0		3.3-V LVTTL

Figure 159. Signal Probe Fitting Results Page in the Compilation Report Window

The **Signal Probe source to output delays** screen in the Timing Analysis section of the Compilation Report displays the timing results of each successfully routed Signal Probe pin.



SignalProbe Source to Output Delays

	Source Name	Pin Location	Pin Name	Enable	Status	Delay (ns)
1	inst5[2]	Pin_H17	probe1	On	Routed	5.135 ns
2	inst5[3]	Pin_H20	probe2	On	Routed	4.939 ns
3	inst5[4]	Pin_G22	probe3	On	Routed	4.475 ns

Figure 160. Signal Probe Source to Output Delays Page in the Compilation Report Window

Note: After a Signal Probe compilation, the processing screen of the Messages window also provides the results for each Signal Probe pin and displays slack information for each successfully routed Signal Probe pin.

13.1.8.1 Analyzing Signal Probe Routing Failures

A Signal Probe compilation can fail for any of the following reasons:

- **Route unavailable**—the Signal Probe compilation failed to find a route from the Signal Probe source to the Signal Probe pin because of routing congestion.
- **Invalid or nonexistent Signal Probe source**—you entered a Signal Probe source that does not exist or is invalid.
- **Unusable output pin**—the output pin selected is found to be unusable.

Routing failures can occur if the Signal Probe pin's I/O standard conflicts with other I/O standards in the same I/O bank.

If routing congestion prevents a successful Signal Probe compilation, you can allow the compiler to modify routing to the specified Signal Probe source. On the Tools menu, click **Signal Probe Pins** and turn on **Modify latest fitting results during Signal Probe compilation**. This setting allows the Fitter to modify existing routing channels used by your design.

Note: Turning on **Modify latest fitting results during Signal Probe compilation** can change the performance of your design.

13.2 Scripting Support

You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Intel Quartus Prime command-line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp
```

Note: The Tcl commands in this section are part of the `::quartus::chip_planner` Intel Quartus Prime Tcl API. Source or include the `::quartus::chip_planner` Tcl package in your scripts to make these commands available.

Related Links

- Tcl Scripting documentation
- Intel Quartus Prime Settings File Reference Manual
  Information about all settings and constraints in the Intel Quartus Prime software
- Command-Line Scripting documentation

13.2.1 Making a Signal Probe Pin

To make a Signal Probe pin, type the following command:

```
make_sp [-h | -help] [-long_help] [-clk <clk>] [-io_std <io_std>] \
-loc <loc> -pin_name <pin name> [-regs <regs>] [-reset <reset>] \
-src_name <source name>
```

13.2.2 Deleting a Signal Probe Pin

To delete a Signal Probe pin, type the following Tcl command:

```
delete_sp [-h | -help] [-long_help] -pin_name <pin name>
```

13.2.3 Enabling a Signal Probe Pin

To enable a Signal Probe pin, type the following Tcl command:

```
enable_sp [-h | -help] [-long_help] -pin_name <pin name>
```

13.2.4 Disabling a Signal Probe Pin

To disable a Signal Probe pin, type the following Tcl command:

```
disable_sp [-h | -help] [-long_help] -pin_name <pin name>
```

13.2.5 Performing a Signal Probe Compilation

To perform a Signal Probe compilation, type the following command:

```
quartus_sh --flow signalprobe <project name>
```

13.2.5.1 Script Example

The example shows a script that creates a Signal Probe pin called `sp1` and connects the `sp1` pin to source node `reg1` in a project that was already compiled.

Creating a Signal Probe Pin Called sp1

```
package require ::quartus::chip_planner
project_open project
read_netlist
make_sp -pin_name sp1 -src_name reg1
check_netlist_and_save
project_close
```

13.2.6 Reserving Signal Probe Pins

To reserve a Signal Probe pin, add the commands shown in the example to the Intel Quartus Prime Settings File (`.qsf`) for your project.

Reserving a Signal Probe Pin

```
set_location_assignment <location> -to <Signal Probe pin name>
set_instance_assignment -name RESERVE_PIN \
"AS SIGNALPROBE OUTPUT" -to <Signal Probe pin name>
```

Valid locations are pin location names, such as `Pin_A3`.

13.2.6.1 Common Problems When Reserving a Signal Probe Pin

If you cannot reserve a Signal Probe pin in the Intel Quartus Prime software, it is likely that one of the following is true:

- You have selected multiple pins.
- A compilation is running in the background. Wait until the compilation is complete before reserving the pin.
- You have the Intel Quartus Prime Lite Edition software, in which the Signal Probe feature is not enabled by default.
- You have not set the pin reserve type to **As Signal Probe Output**. To reserve a pin, on the Assignments menu, in the **Assign Pins** dialog box, select **As Signal Probe Output**.
- The pin is reserved from a previous compilation. During a compilation, the Intel Quartus Prime software reserves each pin on the targeted device. If you end the Intel Quartus Prime process during a compilation, for example, with the **Windows Task Manager End Process** command or the UNIX `kill` command, perform a full recompilation before reserving pins as Signal Probe outputs.
- The pin does not support the Signal Probe feature. Select another pin.
- The current device family does not support the Signal Probe feature.

13.2.7 Adding Signal Probe Sources

To assign the node name to a Signal Probe pin, type the following Tcl command:

```
set_instance_assignment -name SIGNALPROBE_SOURCE <node name> \
-to <signalprobe pin name>
```

The next command turns on Signal Probe routing. To turn off individual Signal Probe pins, specify `OFF` instead of `ON` with the following command:

```
set_instance_assignment -name SIGNALPROBE_ENABLE ON \
-to <Signal Probe pin name>
```

Related Links

- Signal Probe Pins Dialog Box online help
- Add Signal Probe Pins Dialog Box online help
  Information about how to pipeline an existing Signal Probe connection

13.2.8 Assigning I/O Standards

To assign an I/O standard to a pin, type the following Tcl command:

```
set_instance_assignment -name IO_STANDARD <I/O standard> -to <Signal Probe
pin name>
```

Related Links

I/O Standards online help

13.2.9 Adding Registers for Pipelining

To add registers for pipelining, type the following Tcl command:

```
set_instance_assignment -name SIGNALPROBE_CLOCK <clock name> \
-to <Signal Probe pin name>
```

```
set_instance_assignment -name SIGNALPROBE_NUM_REGISTERS <number of registers>
\
-to <Signal Probe pin name>
```

13.2.10 Running Signal Probe Immediately After a Full Compilation

To run Signal Probe immediately after a full compilation, type the following Tcl command:

```
set_global_assignment -name SIGNALPROBE_DURING_NORMAL_COMPILATION ON
```

13.2.11 Running Signal Probe Manually

To run Signal Probe as part of a scripted flow using Tcl, use the following in your script:

```
execute_flow -signalprobe
```

To perform a Signal Probe compilation interactively at a command prompt, type the following command:

```
quartus_sh_fit --flow signalprobe <project name>
```

13.2.12 Enabling or Disabling All Signal Probe Routing

Use the Tcl command in the example to turn on or turn off Signal Probe routing. When using this command, to turn Signal Probe routing on, specify ON. To turn Signal Probe routing off, specify OFF.

Turning Signal Probe On or Off with Tcl Commands

```
set spe [get_all_assignments -name SIGNALPROBE_ENABLE] \
foreach_in_collection asgn $spe {
   set signalprobe_pin_name [lindex $asgn 2]
   set_instance_assignment -name SIGNALPROBE_ENABLE \
-to $signalprobe_pin_name <ON|OFF> }
```

13.2.13 Allowing Signal Probe to Modify Fitting Results

To turn on **Modify latest fitting results**, type the following Tcl command:

```
set_global_assignment -name SIGNALPROBE_ALLOW_OVERUSE ON
```

13.3 Document Revision History

Table 108. Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
June 2014	14.0.0	Dita conversion.

continued...

Date	Version	Changes
May 2013	13.0.0	Changed sequence of flow to clarify that you need to perform a full compilation before reserving Signal Probe pins. Affected sections are "Debugging Using the Signal Probe Feature" on page 12–1 and "Reserving Signal Probe Pins" on page 12–2. Moved "Performing a Full Compilation" on page 12–2 before "Reserving Signal Probe Pins" on page 12–2.
June 2012	12.0.0	Removed survey link.
November 2011	10.0.2	Template update.
December 2010	10.0.1	Changed to new document template.
July 2010	10.0.0	• Revised for new UI. • Removed section Signal Probe ECO flows • Removed support for Signal Probe pin preservation when recompiling with incremental compilation turned on. • Removed outdated FAQ section. • Added links to Intel Quartus Prime Help for procedural content.
November 2009	9.1.0	• Removed all references and procedures for APEX devices. • Style changes.
March 2009	9.0.0	• Removed the "Generate the Programming File" section • Removed unnecessary screenshots • Minor editorial updates
November 2008	8.1.0	• Modified description for preserving Signal Probe connections when using Incremental Compilation • Added plausible scenarios where Signal Probe connections are not reserved in the design
May 2008	8.0.0	• Added "Arria GX" to the list of supported devices • Removed the "On-Chip Debugging Tool Comparison" and replaced with a reference to the Section V Overview on page 13–1 • Added hyperlinks to referenced documents throughout the chapter • Minor editorial updates

Related Links

Documentation Archive
For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

14 Design Debugging with the Signal Tap Logic Analyzer

14.1 About the Signal Tap Logic Analyzer

The Signal Tap Logic Analyzer is a next-generation, system-level debugging tool that captures and displays real-time signal behavior in an FPGA design. You can examine the behavior of internal signals without using extra I/O pins, while the design is running at full speed on an FPGA.

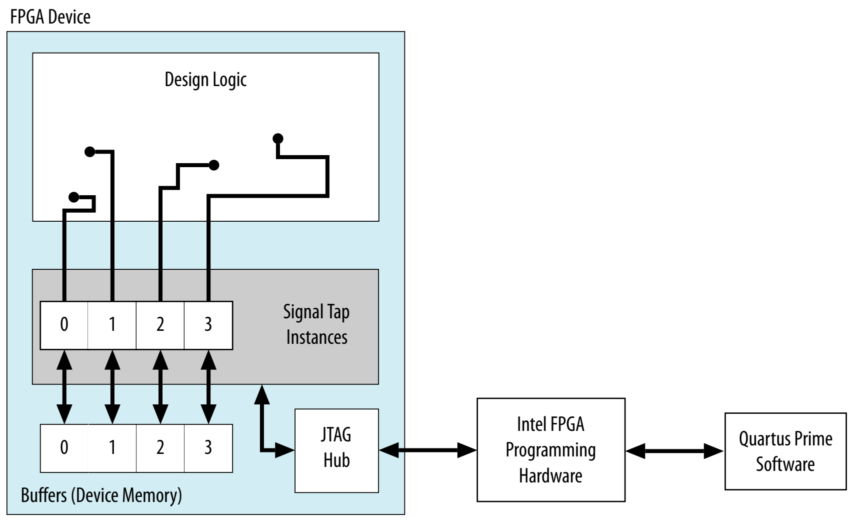
The Signal Tap Logic Analyzer is scalable, easy to use, and available as a stand-alone package or with a software subscription.

The Signal Tap Logic Analyzer supports these features:

- Debug an FPGA design by probing the state of internal signals without the need of external equipment.
- Define custom trigger-condition logic for greater accuracy and improved ability to isolate problems.
- Capture the state of internal nodes or I/O pins in the design without the need of design file changes.
- Store all captured signal data in device memory until you are ready to read and analyze it.

The Signal Tap Logic Analyzer supports the highest number of channels, largest sample depth, and fastest clock speeds of any logic analyzer in the programmable logic market.

Figure 161. Signal Tap Logic Analyzer Block Diagram



Note to figure:

1. This diagram assumes that you compiled the Signal Tap Logic Analyzer with the design as a separate design partition using the Intel Quartus Prime incremental compilation feature. If you do not use incremental compilation, the Compiler integrates the Signal Tap logic with the design.

This chapter is intended for any designer who wants to debug an FPGA design during normal device operation without the need for external lab equipment. Because the Signal Tap Logic Analyzer is similar to traditional external logic analyzers, familiarity with external logic analyzer operations is helpful, but not necessary. To take advantage of faster compile times when making changes to the Signal Tap Logic Analyzer, knowledge of the Intel Quartus Prime incremental compilation feature is helpful.

14.1.1 Hardware and Software Requirements

You need the following hardware and software to perform logic analysis with the Signal Tap Logic Analyzer:

- Signal Tap Logic Analyzer software
- Download/upload cable
- Intel development kit or your design board with JTAG connection to device under test

You can use the Signal Tap Logic Analyzer that is included with the following software:

- Intel Quartus Prime design software
- Intel Quartus Prime Lite Edition

Alternatively, use the Signal Tap Logic Analyzer standalone software and standalone Programmer software.

Note: The Intel Quartus Prime Lite Edition software does not support incremental compilation integration with the Signal Tap Logic Analyzer.

The memory blocks of the device store captured data. The memory blocks transfer the data to the Intel Quartus Prime software waveform display over a JTAG communication cable, such as or Intel FPGA Download Cable.

Table 109. Signal Tap Logic Analyzer Features and Benefits

Feature	Benefit
Quick access toolbar	Provides single-click operation of commonly-used menu items. You can hover over the icons to see tool tips.
Multiple logic analyzers in a single device	Allows you to capture data from multiple clock domains in a design at the same time.
Multiple logic analyzers in multiple devices in a single JTAG chain	Allows you to capture data simultaneously from multiple devices in a JTAG chain.
Nios II plug-in support	Allows you to specify nodes, triggers, and signal mnemonics for IP, such as the Nios II processor.
Up to 10 basic, comparison, or advanced trigger conditions for each analyzer instance	Allows you to send complex data capture commands to the logic analyzer, providing greater accuracy and problem isolation.
Power-up trigger	Captures signal data for triggers that occur after device programming, but before manually starting the logic analyzer.
Custom trigger HDL object	You can code your own trigger in Verilog HDL or VHDL and tap specific instances of modules located anywhere in the hierarchy of your design, without needing to manually route all the necessary connections. This simplifies the process of tapping nodes spread out across your design.
State-based triggering flow	Enables you to organize your triggering conditions to precisely define what your logic analyzer captures.
Incremental compilation	Allows you to modify the signals and triggers that the Signal Tap Logic Analyzer monitors without performing a full compilation, saving time.
Incremental route with rapid recompile	Allows you to manually allocate trigger input, data input, storage qualifier input, and node count, and perform a full compilation to include the Signal Tap Logic Analyzer in your design. Then, you can selectively connect, disconnect, and swap to different nodes in your design. Use Rapid Recompile to perform incremental routing and gain a 2-4x speedup over the initial full compilation.
Flexible buffer acquisition modes	The buffer acquisition control allows you to precisely control the data that is written into the acquisition buffer. Both segmented buffers and non-segmented buffers with storage qualification allow you to discard data samples that are not relevant to the debugging of your design.
MATLAB integration with included MEX function	Collects the data the Signal Tap Logic Analyzer captures into a MATLAB integer matrix.
Up to 2,048 channels per logic analyzer instance	Samples many signals and wide bus structures.
Up to 128K samples per instance	Captures a large sample set for each channel.
Fast clock frequencies	Synchronous sampling of data nodes using the same clock tree driving the logic under test.
Resource usage estimator	Provides an estimate of logic and memory device resources that the Signal Tap Logic Analyzer configurations use.

continued...

Feature	Benefit
No additional cost	Intel Quartus Prime subscription and the Intel Quartus Prime Lite Edition include the Signal Tap Logic Analyzer.
Compatibility with other on-chip debugging utilities	You can use the Signal Tap Logic Analyzer in tandem with any JTAG-based on-chip debugging tool, such as an In-System Memory Content editor, allowing you to change signal values in real-time while you are running an analysis with the Signal Tap Logic Analyzer.
Floating-Point Display Format	To enable, click **Edit ➤ Bus Display Format ➤ Floating-point** Supports: • Single-precision floating-point format **IEEE754 Single (32-bit)**. • Double-precision floating-point format **IEEE754 Double (64-bit)**.

Related Links

System Debugging Tools Overview on page 183

14.1.2 Open Standalone Signal Tap Logic Analyzer GUI

To open a new Signal Tap through the command-line, type:

```
quartus_stpw <stp_file.stp>
```

14.1.3 Backward Compatibility with Previous Versions of Intel Quartus Prime Software

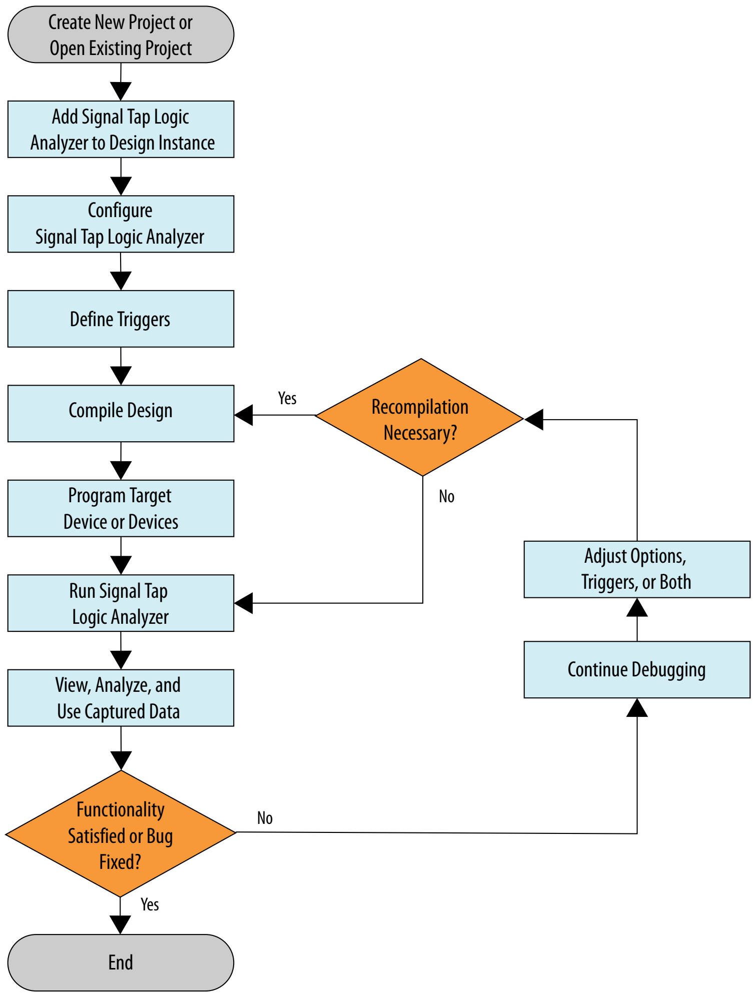
When you open an `.stp` file created in a previous version of Intel Quartus Prime software in a newer version of the software, the `.stp` file cannot be opened in a previous version of the Intel Quartus Prime software.

If you have a Intel Quartus Prime project file from a previous version of the software, you may have to update the `.stp` configuration file to recompile the project. You can update the configuration file by opening the Signal Tap Logic Analyzer. If you need to update your configuration, a prompt appears asking if you want to update the `.stp` to match the current version of the Intel Quartus Prime software.

14.2 Signal Tap Logic Analyzer Task Flow Overview

To use the Signal Tap Logic Analyzer to debug your design, you perform a number of tasks to add, configure, and run the logic analyzer.

Figure 162. Signal Tap Logic Analyzer Task Flow


Create New Project or Open Existing Project
Add Signal Tap Logic Analyzer to Design Instance
Configure Signal Tap Logic Analyzer
Define Triggers
Compile Design
Yes
Recompilation Necessary?
No
Program Target Device or Devices
Adjust Options, Triggers, or Both
Run Signal Tap Logic Analyzer
Continue Debugging
View, Analyze, and Use Captured Data
Functionality Satisfied or Bug Fixed?
No
Yes
End

14.2.1 Add the Signal Tap Logic Analyzer to Your Design

Create an `.stp` or create a parameterized HDL instance representation of the logic analyzer using the IP Catalog and parameter editor. If you want to monitor multiple clock domains simultaneously, add additional instances of the logic analyzer to your design, limited only by the available resources in your device.

14.2.2 Configure the Signal Tap Logic Analyzer

After you add the Signal Tap Logic Analyzer to your design, configure the logic analyzer to monitor the signals you want.

You can add signals manually or use a plug-in, such as the Nios II processor plug-in, to add entire sets of associated signals for a particular IP.

Specify settings for the data capture buffer, such as its size, the method in which the Signal Tap Logic Analyzer captures and stores the data. If your device supports memory type selection, you can specify the memory type to use for the buffer.

Related Links

Configuring the Signal Tap Logic Analyzer on page 333

14.2.3 Define Trigger Conditions

To capture and store specific signal data, set up triggers that tell the logic analyzer under what conditions to stop capturing data. The Signal Tap Logic Analyzer captures data continuously while the logic analyzer is running.

The Signal Tap Logic Analyzer allows you to define trigger conditions that range from very simple, such as the rising edge of a single signal, to very complex, involving groups of signals, extra logic, and multiple conditions. Power-Up Triggers allow you to capture data from trigger events occurring immediately after the device enters user-mode after configuration.

Related Links

Defining Triggers on page 352

14.2.4 Compile the Design

Once you configure the `.stp` file and define trigger conditions, compile your project including the logic analyzer in your design.

Note: Because you may need to change monitored signal nodes or adjust trigger settings frequently during debugging, Intel FPGA recommends that you use the incremental compilation feature built into the Signal Tap Logic Analyzer, along with Intel Quartus Prime incremental compilation, to reduce recompile times. You can also use Incremental Route with Rapid Recompile to reduce recompile times.

Related Links

Compiling the Design on page 376

14.2.5 Program the Target Device or Devices

When you debug a design with the Signal Tap Logic Analyzer, you can program a target device directly from the `.stp` without using the Intel Quartus Prime Programmer. You can also program multiple devices with different designs and simultaneously debug them.

Related Links

- Program the Target Device or Devices on page 381
- Manage Multiple Signal Tap Files and Configurations on page 350

14.2.6 Run the Signal Tap Logic Analyzer

In normal device operation, you control the logic analyzer through the JTAG connection, specifying when to start looking for trigger conditions to begin capturing data. With Runtime or Power-Up Triggers, read and transfer the captured data from the on-chip buffer to the `.stp` for analysis.

Related Links

Running the Signal Tap Logic Analyzer on page 382

14.2.7 View, Analyze, and Use Captured Data

The data you capture and read into the `.stp` file is available for analysis and debugging. You can save the data for later analysis, or convert the data to other formats for sharing and further study.

- To simplify reading and interpreting the signal data you capture, set up mnemonic tables, either manually or with a plug-in.
- To speed up debugging, use the **Locate** feature in the **Signal Tap node** list to find the locations of problem nodes in other tools in the Intel Quartus Prime software.

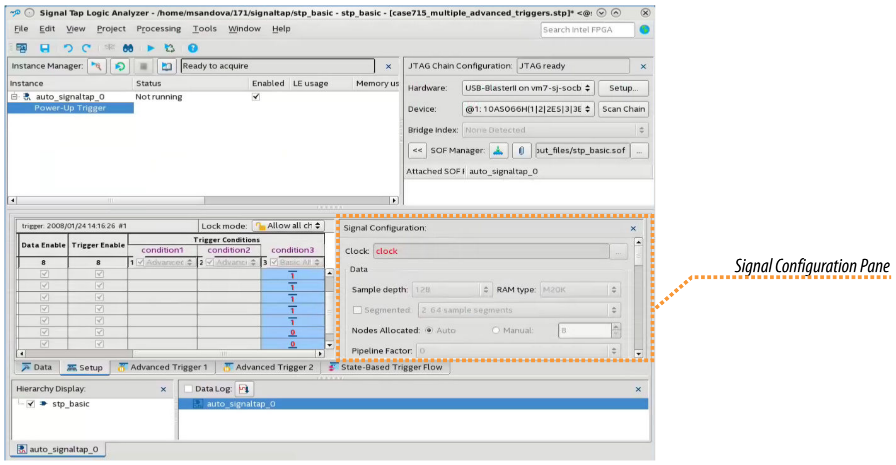
Related Links

View, Analyze, and Use Captured Data on page 386

14.3 Configuring the Signal Tap Logic Analyzer

You can configure instances of the Signal Tap Logic Analyzer in the **Signal Configuration** pane of the **Signal Tap Logic Analyzer** window. Some settings are similar to those found on traditional external logic analyzers. Other settings are unique to the Signal Tap Logic Analyzer.

Figure 163. Signal Tap Logic Analyzer Signal Configuration Pane



Note: You can adjust fewer settings with run-time trigger conditions than with power-up trigger conditions.

14.3.1 Assigning an Acquisition Clock

To control how the Signal Tap Logic Analyzer acquires data you must assign a clock signal. The logic analyzer samples data on every positive (rising) edge of the acquisition clock. The logic analyzer does not support sampling on the negative (falling) edge of the acquisition clock.

You can use any signal in your design as the acquisition clock. However, for best results in data acquisition, use a global, non-gated clock that is synchronous to the signals under test. Using a gated clock as your acquisition clock can result in unexpected data that does not accurately reflect the behavior of your design. The Intel

Quartus Prime static timing analysis tools show the maximum acquisition clock frequency at which you can run your design. To find the maximum frequency of the logic analyzer clock, refer to the Timing Analysis section of the Compilation Report.

Caution: Be careful when using a recovered clock from a transceiver as an acquisition clock for the Signal Tap Logic Analyzer. A recovered clock can cause incorrect or unexpected behavior, particularly when the transceiver recovered clock is the acquisition clock with the power-up trigger feature.

If you do not assign an acquisition clock in the Signal Tap Logic Analyzer Editor, Intel Quartus Prime software automatically creates a clock pin called `auto_stp_external_clk`. You must make a pin assignment to this pin, and make sure that a clock signal in your design drives the acquisition clock.

Related Links

- Adding Signals with a Plug-In on page 337
- Managing Device I/O Pins
  In *Intel Quartus Prime Standard Edition Handbook Volume 2*

14.3.2 Adding Signals to the Signal Tap File

Add the signals that you want to monitor to the `.stp` node list. You can also select signals to define triggers. You can assign the following two signal types:

- **Pre-synthesis**—These signals exist after design elaboration, but before any synthesis optimizations are done. This set of signals must reflect your Register Transfer Level (RTL) signals.
- **Post-fitting**—These signals exist after physical synthesis optimizations and place-and-route.

Note: If you are not using incremental compilation, add only pre-synthesis signals to the `.stp`. Using pre-synthesis helps when you want to add a new node after you change a design. After you perform Analysis and Elaboration, the source file changes appear in the Node Finder.

Intel Quartus Prime software does not limit the number of signals available for monitoring in the Signal Tap window waveform display. However, the number of channels available is directly proportional to the number of logic elements (LEs) or adaptive logic modules (ALMs) in the device. Therefore, there is a physical restriction on the number of channels that are available for monitoring. Signals shown in blue text are post-fit node names. Signals shown in black text are pre-synthesis node names.

After successful Analysis and Elaboration, invalid signals appear in red. Unless you are certain that these signals are valid, remove them from the `.stp` file for correct operation. The Signal Tap Status Indicator also indicates if an invalid node name exists in the `.stp` file.

You can tap signals if a routing resource (row or column interconnects) exists to route the connection to the Signal Tap instance. For example, you cannot tap signals that exist in the I/O element (IOE), because there are no direct routing resources from the signal in an IOE to a core logic element. For input pins, you can tap the signal that is driving a logic array block (LAB) from an IOE, or, for output pins, you can tap the signal from the LAB that is driving an IOE.

Related Links

- Faster Compilations with Intel Quartus Prime Incremental Compilation on page 376
- Setup Tab (Signal Tap Logic Analyzer)
  In *Intel Quartus Prime Help*

14.3.2.1 About Adding Pre-Synthesis Signals

When you add pre-synthesis signals, make all connections to the Signal Tap Logic Analyzer before synthesis. The Compiler allocates logic and routing resources to make the connection as if you changed your design files. For signals driving to and from IOEs, pre-synthesis signal names coincide with the pin's signal names.

14.3.2.2 About Adding Post-Fit Signals

In the case of post-fit signals, connections that you make to the Signal Tap Logic Analyzer are the signal names from the actual atoms in your post-fit netlist. You can only make a connection if the signals are part of the existing post-fit netlist, and existing routing resources are available from the signal of interest to the Signal Tap Logic Analyzer.

In the case of post-fit output signals, tap the `COMBOUT` or `REGOUT` signal that drives the IOE block. For post-fit input signals, signals driving into the core logic coincide with the pin's signal name.

Note: Because `NOT`-gate push back applies to any register that you tap, the signal from the atom may be inverted. You can check this by locating the signal in either the Resource Property Editor or the Technology Map Viewer. You can also use the Technology Map viewer and the Resource Property Editor to find post-fit node names.

Related Links

Design Flow with the Netlist Viewers
In *Intel Quartus Prime Standard Edition Handbook Volume 1*

14.3.2.2.1 Assigning Data Signals Using the Technology Map Viewer

You can use the Technology Map Viewer to add post-fit signal names easily. To do so, launch the Technology Map Viewer (post-fitting) after compilation. When you find the desired node, copy the node to either the active `.stp` for your design or a new `.stp`.

To launch the Technology Map Viewer, click **Tools ➤ Netlist Viewers ➤ Technology Map Viewer (Post-Fitting)** in the **Intel Quartus Prime** window.

14.3.2.3 Preserving Signals

The Intel Quartus Prime software optimizes the RTL signals during synthesis and place-and-route. RTL signal names may not appear in the post-fit netlist after optimizations. For example, the compilation process can add tildes (~) to nets that fan-out from a node, making it difficult to decipher which signal nets they actually represent.

The Intel Quartus Prime software provides synthesis attributes that prevent the Compiler to perform any optimization on the specified signals, allowing them to persist into the post-fit netlist:

- `keep`—Prevents removal of combinational signals during optimization.
- `preserve`—Prevents removal of registers during optimization.

However, using preserving attributes can increase device resource utilization or decrease timing performance.

Note: These processing results can cause problems when you use the incremental compilation flow with the Signal Tap Logic Analyzer. Because you can only add post-fitting signals to the Signal Tap Logic Analyzer in partitions of type **post-fit**, RTL signals that you want to monitor may not be available, preventing their use. To avoid this issue, use synthesis attributes to preserve signals during synthesis and place-and-route.

If you are debugging an IP core, such as the Nios II CPU or other encrypted IP, you might need to preserve nodes from the core to keep available for debugging with the Signal Tap Logic Analyzer. Preserving nodes is often necessary when you use a plug-in to add a group of signals for a particular IP.

If you use incremental compilation flow with the Signal Tap Logic Analyzer, pre-synthesis nodes may not be connected to the Signal Tap Logic Analyzer if the affected partition is of the post-fit type. Signal Tap issues a critical warning for all pre-synthesis node names that it does not find in the post-fit netlist.

14.3.2.4 Node List Signal Use Options

When you add a signal to the node list, you can select options that specify how the logic analyzer uses the signal.

To prevent a signal from triggering the analysis, disable the signal's **Trigger Enable** option in the `.stp` file. This option is useful when you only want to see the signal's captured data.

You can turn off the ability to view data for a signal by disabling the **Data Enable** column in the `.stp` file. This option is useful when you want to trigger on a signal, but have no interest in viewing that signal's data.

Related Links

Defining Triggers on page 352

14.3.2.4.1 Disabling and Enabling a Signal Tap Instance

Disable and enable Signal Tap instances in the **Instance Manager** pane. Physically adding or removing instances requires recompilation after disabling and enabling a Signal Tap instance.

14.3.2.5 Untappable Signals

Not all the post-fitting signals in your design are available in the **Signal Tap : post-fitting filter** in the **Node Finder** dialog box.

You cannot tap any of the following signal types:

- **Post-fit output pins**—You cannot tap a post-fit output pin directly. To make an output signal visible, tap the register or buffer that drives the output pin. This includes pins defined as bidirectional.
- **Signals that are part of a carry chain**—You cannot tap the carry out (`cout0` or `cout1`) signal of a logic element. Due to architectural restrictions, the carry out signal can only feed the carry in of another LE.
- **JTAG Signals**—You cannot tap the JTAG control (`TCK`, `TDI`, `TDO`, and `TMS`) signals.
- **ALTGXB IP core**—You cannot directly tap any ports of an ALTGXB instantiation.
- **LVDS**—You cannot tap the data output from a serializer/deserializer (SERDES) block.
- **DQ**, **DQS Signals**—You cannot directly tap the `DQ` or `DQS` signals in a DDR/DDRII design.

14.3.3 Adding Signals with a Plug-In

Instead of adding individual or grouped signals through the **Node Finder**, you can use a plug-in to add groups of relevant signals of a particular type of IP. Besides easy signal addition, plug-ins provide features such as pre-designed mnemonic tables, useful for trigger creation and data viewing, as well as the ability to disassemble code in captured data. The Signal Tap Logic Analyzer comes with one plug-in for the Nios II processor.

The Nios II plug-in, for example, creates one mnemonic table in the **Setup** tab and two tables in the **Data** tab:

- **Nios II Instruction** (**Setup** tab)—Capture all the required signals for triggering on a selected instruction address.
- **Nios II Instance Address** (**Data** tab)—Display address of executed instructions in hexadecimal format or as a programming symbol name if defined in an optional Executable and Linking Format (`.elf`) file.
- **Nios II Disassembly** (**Data** tab)—Display disassembled code from the corresponding address.

To add signals to the `.stp` file using a plug-in, perform the following steps after running Analysis and Elaboration on your design:

1. To ensure that all the required signals are available, in the Intel Quartus Prime software, click **Assignments ➤ Settings ➤ Compiler Settings ➤ Advanced Settings (Synthesis)**. Turn on **Create debugging nodes for IP cores**.
   All the signals included in the plug-in are added to the node list.
2. Right-click the node list. On the **Add Nodes with Plug-In** submenu, select the plug-in you want to use, such as the included plug-in named **Nios II**.
   The **Select Hierarchy Level** dialog box appears showing the IP hierarchy of your design. If the IP for the selected plug-in does not exist in your design, a message informs you that you cannot use the selected plug-in.
3. Select the IP that contains the signals you want to monitor with the plug-in, and click **OK**.

— If all the signals in the plug-in are available, a dialog box might appear, depending on the plug-in, where you can specify options for the plug-in.

4. With the Nios II plug-in, you can optionally select an `.elf` containing program symbols from your Nios II Integrated Development Environment (IDE) software design. Specify options for the selected plug-in, and click **OK**.

Related Links

- Defining Triggers on page 352
- View, Analyze, and Use Captured Data on page 333

14.3.4 Adding Finite State Machine State Encoding Registers

Finding the signals to debug finite state machines (FSM) can be challenging. Finding nodes from the post-fit netlist may be impossible, since the Compiler may change or optimize away FSM encoding signals. To find and map FSM signal values to the state names that you specified in your HDL, you must perform an additional step.

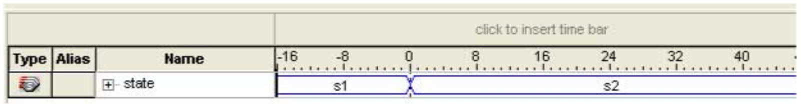
The Signal Tap Logic Analyzer can detect FSMs in your compiled design. The configuration automatically tracks the FSM state signals as well as state encoding through the compilation process.

To add all the FSM state signals to your logic analyzer with a single command Shortcut menu commands allow you .

For each FSM added to your Signal Tap configuration, the FSM debugging feature adds a mnemonic table to map the signal values to the state enumeration that you provided in your source code. The mnemonic tables enable you to visualize state machine transitions in the waveform viewer. The FSM debugging feature supports adding FSM signals from both the pre-synthesis and post-fit netlists.

Figure 164. Decoded FSM Mnemonics

The waveform viewer with decoded signal values from a state machine added with the FSM debugging feature.



Related Links

State Machine HDL Guidelines
In *Intel Quartus Prime Standard Edition Handbook Volume 1*

14.3.4.1 Modify and Restore Mnemonic Tables for State Machines

Edit any mnemonic table using the **Mnemonic Table Setup** dialog box. When you add FSM state signals via the FSM debugging feature, the Signal Tap Logic Analyzer GUI creates a mnemonic table using the format <*StateSignalName*>`_table`, where **StateSignalName** is the name of the state signals that you have declared in your RTL.

If you want to restore a mnemonic table that was modified, right-click anywhere in the node list window and select **Recreate State Machine Mnemonics**. By default, restoring a mnemonic table overwrites the existing mnemonic table that you modified. To restore a FSM mnemonic table to a new record, turn off **Overwrite existing mnemonic table** in the **Recreate State Machine Mnemonics** dialog box.

Note: If you have added or deleted a signal from the FSM state signal group from within the setup tab, delete the modified register group and add the FSM signals back again.

Related Links

Creating Mnemonics for Bit Patterns on page 389

14.3.4.2 Additional Considerations for State Machines in Signal Tap

- The Signal Tap configuration GUI recognizes state machines from your design only if you use Intel Quartus Prime Integrated Synthesis. Conversely, the state machine debugging feature is not able to track the FSM signals or state encoding if you use other EDA synthesis tools.
- If you add post-fit FSM signals, the Signal Tap Logic Analyzer FSM debug feature may not track all optimization changes that are a part of the compilation process.
- If the following two specific optimizations are enabled, the Signal Tap FSM debug feature may not list mnemonic tables for state machines in the design:
  - If you enabled the **Physical Synthesis** optimization, state registers may be resource balanced (register retiming) to improve f_{MAX}. The FSM debug feature does not list post-fit FSM state registers if register retiming occurs.
  - The FSM debugging feature does not list state signals that the Compiler packed into RAM and DSP blocks during synthesis or Fitter optimizations.
- You can still use the FSM debugging feature to add pre-synthesis state signals.

Related Links

Enabling Physical Synthesis Optimization
In *Intel Quartus Prime Standard Edition Handbook Volume 1*

14.3.5 Specify the Sample Depth

The **Sample depth** setting specifies the number of samples the Signal Tap Logic Analyzer captures and stores, for each signal in the captured data buffer. To specify the sample depth, select the desired number in the **Sample Depth** drop-down menu. The sample depth ranges from 0 to 128K.

If device memory resources are limited, you may not be able to successfully compile your design with the sample buffer size you have selected. Try reducing the sample depth to reduce resource usage.

Related Links

Signal Configuration Pane (View Menu) (Signal Tap Logic Analyzer)
In *Intel Quartus Prime Help*

14.3.6 Capture Data to a Specific RAM Type

You have the option to select the RAM type where the Signal Tap Logic Analyzer stores acquisition data. Once you allocate the Signal Tap Logic Analyzer buffer to a particular RAM block, the entire RAM block becomes a dedicated resource for the logic analyzer.

RAM selection allows you to preserve a specific memory block for your design, and allocate another portion of memory for Signal Tap Logic Analyzer data acquisition.

For example, if your design has an application that requires a large block of memory resources, such as a large instruction or data cache, you can use MLAB, M512, or M4k blocks for data acquisition and leave M9k blocks for the rest of your design.

To specify the RAM type to use for the Signal Tap Logic Analyzer buffer, go to the **Signal Configuration** pane in the **Signal Tap** window, and select one **Ram type** from the drop-down menu.

Use this feature only when the acquired data is smaller than the available memory of the RAM type that you selected. The amount of data appears in the Signal Tap resource estimator.

Related Links

Signal Configuration Pane (View Menu) (Signal Tap Logic Analyzer)
In *Intel Quartus Prime Help*

14.3.7 Select the Buffer Acquisition Mode

When you specify how the logic analyzer organizes the captured data buffer, you can potentially reduce the amount of memory that Signal Tap requires for data acquisition.

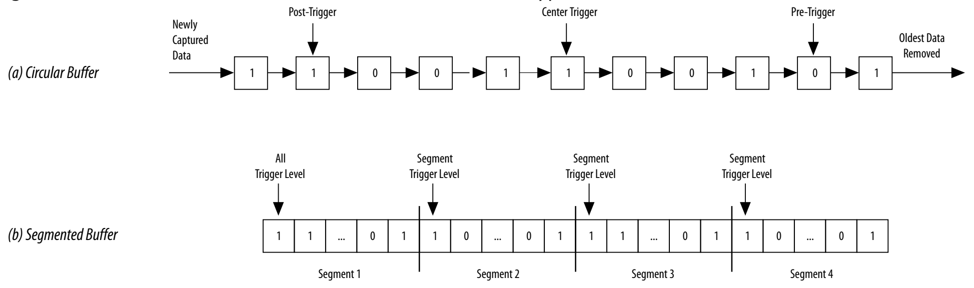
There are two types of acquisition buffer within the Signal Tap Logic Analyzer—a non-segmented (or circular) buffer and a segmented buffer.

- With a non-segmented buffer, the Signal Tap Logic Analyzer treats entire memory space as a single FIFO, continuously filling the buffer until the logic analyzer reaches a defined set of trigger conditions.
- With a segmented buffer, the memory space is split into separate buffers. Each buffer acts as a separate FIFO with its own set of trigger conditions, and behaves as a non-segmented buffer. Only a single buffer is active during an acquisition. The Signal Tap Logic Analyzer advances to the next segment after the trigger condition or conditions for the active segment has been reached.

When using a non-segmented buffer, you can use the storage qualification feature to determine which samples are written into the acquisition buffer. Both the segmented buffers and the non-segmented buffer with the storage qualification feature help you maximize the use of the available memory space.

Figure 165. Buffer Type Comparison in the Signal Tap Logic Analyzer

The figure illustrates the differences between the two buffer types.



Both non-segmented and segmented buffers can use a preset trigger position (Pre-Trigger, Center Trigger, Post-Trigger). Alternatively, you can define a custom trigger position using the **State-Based Triggering** tab. Refer to *Specify Trigger Position* for more details.

Notes to figure:

Related Links

- Specify Trigger Position on page 372
- Using the Storage Qualifier Feature on page 343

14.3.7.1 Non-Segmented Buffer

The non-segmented buffer is the default buffer type in the Signal Tap Logic Analyzer.

At runtime, the logic analyzer stores data in the buffer until the buffer fills up. From that point on, new data overwrites the oldest data, until a specific trigger event occurs. The amount of data the buffer captures after the trigger event depends on the **Trigger position** setting:

- To capture most data before the trigger occurs, select **Post trigger position** from the list
- To capture most data after the trigger, select **Pre trigger position**.
- To center the trigger position in the data, select **Center trigger position**.

Alternatively, use the custom State-based triggering flow to define a custom trigger position within the capture buffer.

Related Links

Specify Trigger Position on page 372

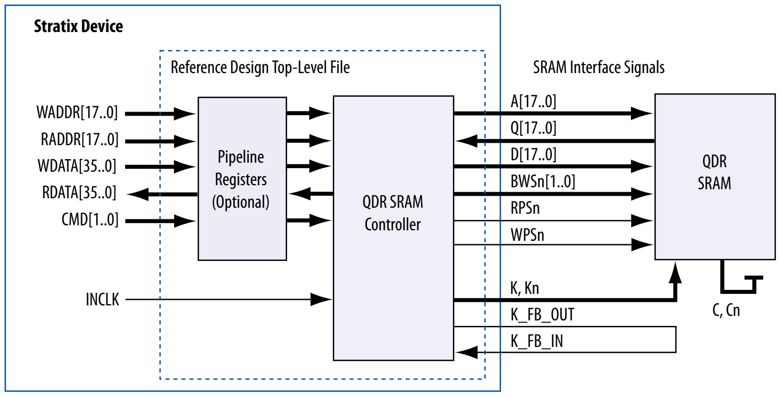
14.3.7.2 Segmented Buffer

In a segmented buffer, the acquisition memory is split into segments of even size, and you define a set of trigger conditions for each segment. Each segment acts as a non-segmented buffer. A segmented buffer allows you to debug systems that contain relatively infrequent recurring events.

If you want to have separate trigger conditions for each of the segmented buffers, you must use the state-based trigger flow. The figure shows an example of a segmented buffer system.

Figure 166. System that Generates Recurring Events

In this design, you want to ensure that the correct data is written to the SRAM controller by monitoring the RDATA port whenever the address `H'0F0F0F0F` is sent into the RADDR port.



With the buffer acquisition feature. you can monitor multiple read transactions from the SRAM device without running the Signal Tap Logic Analyzer again, because you split the memory to capture the same event multiple times, without wasting allocated memory. The buffer captures as many cycles as the number of segments you define under the **Data** settings in the **Signal Configuration** pane.

To enable and configure buffer acquisition, select **Segmented** in the Signal Tap Logic Analyzer Editor and determine the number of segments to use. In the example in the figure, selecting sixty-four 64-sample segments allows you to capture 64 read cycles.

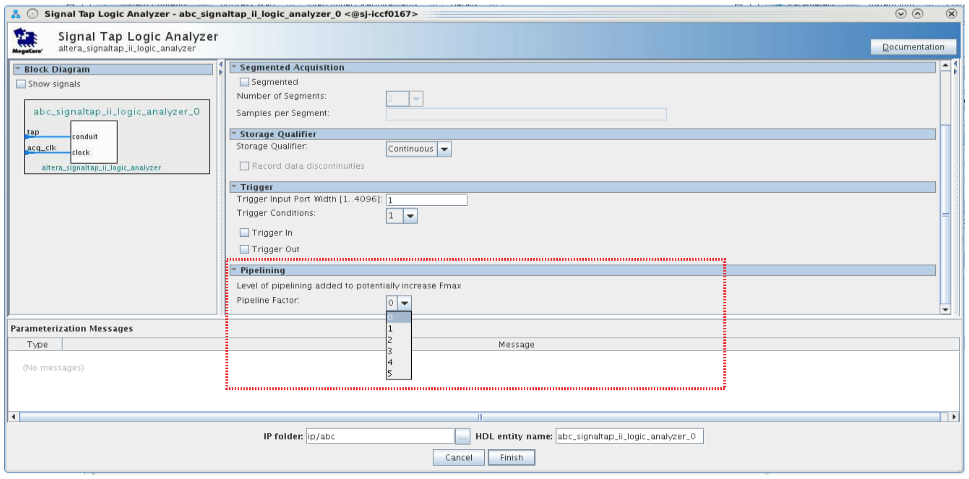
14.3.8 Specify the Pipeline Factor

The **Pipeline factor** setting indicates the number of pipeline registers that you can add to boost the f_{MAX} of the Signal Tap Logic Analyzer. You can specify the pipeline factor in the **Signal Configuration** pane. The pipeline factor ranges from 0 to 5, with a default value of 0.

You can also set the pipeline factor when you instantiate the Signal Tap Logic Analyzer component from your Platform Designer (Standard) system:

1. Double-click **Signal Tap Logic Analyzer** component in the IP Catalog.
2. Specify the **Pipeline Factor**, along with other parameter values.

Figure 167. Specifying the Pipeline Factor from Platform Designer (Standard)


Signal Tap Logic Analyzer - abc_signaltap_ii_logic_analyzer_0 <@sj-iccf0167>
Signal Tap Logic Analyzer
altera_signaltap_ii_logic_analyzer
Documentation
Block Diagram
Show signals
abc_signaltap_ii_logic_analyzer_0
tap
acq_clk
conduit
clock
altera_signaltap_ii_logic_analyzer
Parameterization Messages
Type
Message
(No messages)
Cancel
Finish

Segmented Acquisition
- Segmented: ☐
- Number of Segments: 2
- Samples per Segment:

Storage Qualifier
- Storage Qualifier: Continuous
- Record data discontinuities: ☐

Trigger
- Trigger Input Port Width [1..4096]: 1
- Trigger Conditions: 1
- Trigger In: ☐
- Trigger Out: ☐

Pipelining
- Level of pipelining added to potentially increase Fmax
- Pipeline Factor: 0 (0, 1, 2, 3, 4, 5)

IP folder: ip/abc HDL entity name: abc_signaltap_ii_logic_analyzer_0

Note: Setting the pipeline factor does not guarantee an increase in f_{MAX}, as the pipeline registers may not be in the critical paths.

14.3.9 Using the Storage Qualifier Feature

The Storage Qualifier feature allows you to filter out individual samples not relevant to debugging the design.

The Signal Tap Logic Analyzer offers a snapshot in time of the data stored in the acquisition buffers. By default, the Signal Tap Logic Analyzer writes into acquisition memory with data samples on every clock cycle. With a non-segmented buffer, there is one data window that represents a comprehensive snapshot of the data stream. Conversely, segmented buffers use several smaller sampling windows spread out over more time, with each sampling window representing a contiguous data set.

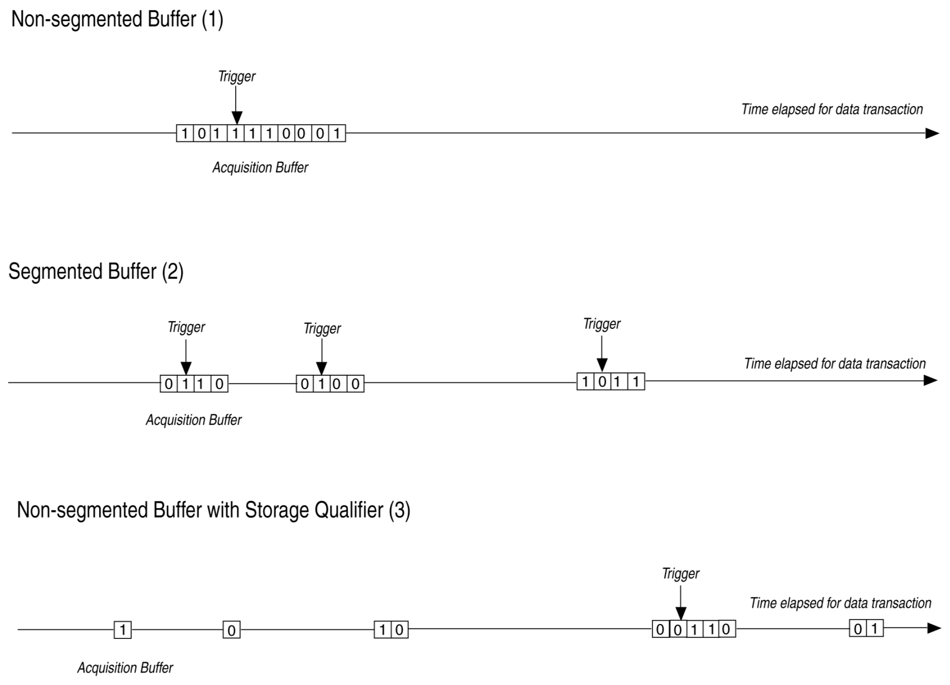
With analysis using acquisition buffers you can capture most functional errors in a chosen signal set, provided adequate trigger conditions and a generous sample depth for the acquisition. However, each data window can have a considerable amount of unnecessary data; for example, long periods of idle signals between data bursts. The default behavior in the Signal Tap Logic Analyzer doesn't discard the redundant sample bits.

The Storage Qualifier feature allows you to establish a condition that acts as a write enable to the buffer during each clock cycle of data acquisition, thus allowing a more efficient use of acquisition memory over a longer period of analysis.

Because you can create a discontinuity between any two samples in the buffer, the Storage Qualifier feature is equivalent to creating a custom segmented buffer in which the number and size of segment boundaries are adjustable.

Note: You can only use the Storage Qualifier feature with a non-segmented buffer. The IP Catalog flow only supports the Input Port mode for the Storage Qualifier feature.

Figure 168. Data Acquisition Using Different Modes of Controlling the Acquisition Buffer



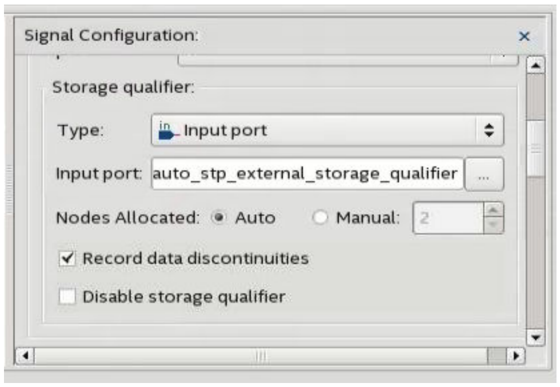
Notes to figure:

1. Non-segmented buffers capture a fixed sample window of contiguous data.
2. Segmented buffers divide the buffer into fixed sized segments, with each segment having an equal sample depth.
3. Storage Qualifier allows you to define a custom sampling window for each segment you create with a qualifying condition, thus potentially allowing a larger time scale of coverage.

There are six storage qualifier types available under the Storage Qualifier feature:

- **Continuous** (default) Turns the Storage Qualifier off.
- **Input port**
- **Transitional**
- **Conditional**
- **Start/Stop**
- **State-based**

Figure 169. Storage Qualifier Settings


Signal Configuration:

Storage qualifier:

Type: Input port

Input port: auto_stp_external_storage_qualifier

Nodes Allocated: [✓ Auto] Manual: 2

[✓] Record data discontinuities

[ ] Disable storage qualifier

Upon the start of an acquisition, the Signal Tap Logic Analyzer examines each clock cycle and writes the data into the buffer based upon the storage qualifier type and condition. Acquisition stops when a defined set of trigger conditions occur.

The Signal Tap Logic Analyzer evaluates trigger conditions independently of storage qualifier conditions.

Related Links

Define Trigger Conditions on page 332

14.3.9.1 Input Port Mode

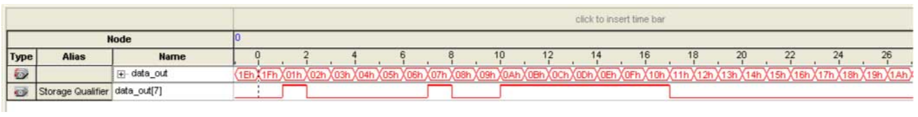
When using the Input port mode, the Signal Tap Logic Analyzer takes any signal from your design as an input. During acquisition, if the signal is high on the clock edge, the Signal Tap Logic Analyzer stores the data in the buffer. If the signal is low on the clock edge, the Logic Analyzer ignores the data sample. If you don't specify an internal node, the Logic Analyzer creates and connects a pin to this input port.

If you are creating a Signal Tap Logic Analyzer instance through an `.stp` file, specify the storage qualifier signal using the input port field located on the **Setup** tab. You must specify this port for your project to compile.

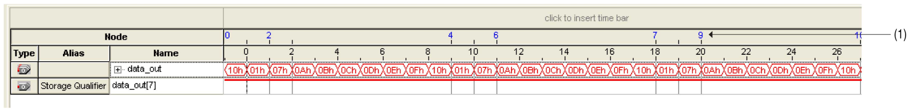
If you use the parameter editor, the storage qualification input port, if specified, appears in the generated instantiation template. You can then connect this port to a signal in your RTL.

Figure 170. Comparing Continuous and Input Port Capture Mode in Data Acquisition of a Recurring Data Pattern

- Continuous Mode:


click to insert time bar
Node
Type
Alias
Name
data_out
Storage Qualifier
data_out[7]
1Eh 1Fh 01h 02h 03h 04h 05h 06h 07h 08h 09h 0Ah 0Bh 0Ch 0Dh 0Eh 0Fh 10h 11h 12h 13h 14h 15h 16h 17h 18h 19h 1Ah

- Input Port Storage Qualifier:

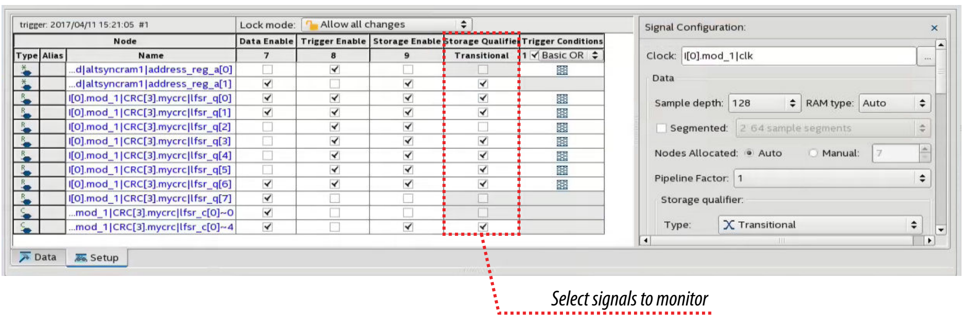

click to insert time bar
Node
Type
Alias
Name
data_out
Storage Qualifier
data_out[7]
10h 01h 07h 0Ah 0Bh 0Ch 0Dh 0Eh 0Fh 10h 01h 07h 0Ah 0Bh 0Ch 0Dh 0Eh 0Fh 10h 01h 07h 0Ah 0Bh 0Ch 0Dh 0Eh 0Fh 10h
(1)

(1) Markers display samples when the logic analyzer paused a write into acquisition memory. These markers are enabled with the option "Record data discontinuities."

14.3.9.2 Transitional Mode

In Transitional mode, the Logic Analyzer monitors changes in a set of signals, and writes new data in the acquisition buffer only when it detects a change. You select the signals for monitoring using the check boxes in the **Storage Qualifier** column.

Figure 171. Transitional Storage Qualifier Setup



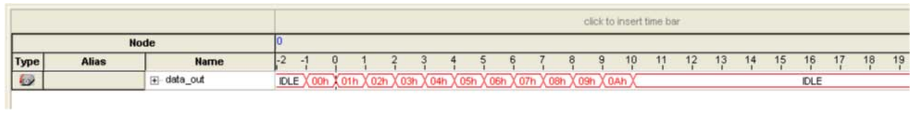
trigger: 2017/04/11 15:21:05 #1 — Lock mode: Allow all changes

Type	Alias	Name	Data Enable 7	Trigger Enable 8	Storage Enable 9	Storage Qualifier Transitional	Trigger Conditions 1 ✓ Basic OR
		...d|altsyncram1|address_reg_a[0]	☐	✓	☐	☐	▨
		...d|altsyncram1|address_reg_a[1]	✓	☐	✓	✓	
		I[0].mod_1|CRC[3].mycrc|lfsr_q[0]	✓	✓	✓	✓	▨
		I[0].mod_1|CRC[3].mycrc|lfsr_q[1]	✓	✓	✓	✓	▨
		I[0].mod_1|CRC[3].mycrc|lfsr_q[2]	☐	✓	✓	☐	▨
		I[0].mod_1|CRC[3].mycrc|lfsr_q[3]	☐	✓	✓	✓	▨
		I[0].mod_1|CRC[3].mycrc|lfsr_q[4]	☐	✓	✓	✓	▨
		I[0].mod_1|CRC[3].mycrc|lfsr_q[5]	☐	✓	✓	✓	▨
		I[0].mod_1|CRC[3].mycrc|lfsr_q[6]	✓	✓	✓	✓	▨
		I[0].mod_1|CRC[3].mycrc|lfsr_q[7]	✓	☐	☐	☐	
		...mod_1|CRC[3].mycrc|lfsr_c[0]~0	✓	☐	☐	☐	
		...mod_1|CRC[3].mycrc|lfsr_c[0]~4	✓	☐	✓	✓	

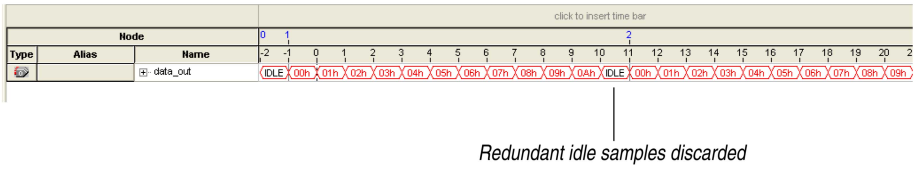
Signal Configuration:
Clock: I[0].mod_1|clk
Data
Sample depth: 128 RAM type: Auto
☐ Segmented: 2 64 sample segments
Nodes Allocated: [✓ Auto] Manual: 7
Pipeline Factor: 1
Storage qualifier:
Type: Transitional

Figure 172. Comparing Continuous and Transitional Capture Mode in Data Acquisition of a Recurring Data Pattern

- Continuous:



- Transitional mode:


click to insert time bar
Node
Type
Alias
Name
data_out
IDLE 00h 01h 02h 03h 04h 05h 06h 07h 08h 09h 0Ah IDLE 00h 01h 02h 03h 04h 05h 06h 07h 08h 09h
Redundant idle samples discarded

14.3.9.3 Conditional Mode

In Conditional mode, the Signal Tap Logic Analyzer determines whether to store a sample by evaluating a combinational function of predefined signals within the node list. The Signal Tap Logic Analyzer writes into the buffer during the clock cycles in which the condition you specify evaluates TRUE.

You can select either **Basic AND**, **Basic OR**, **Comparison**, or **Advanced** storage qualifier conditions. A **Basic AND** or **Basic OR** condition matches each signal to one of the following:

- **Don’t Care**
- **Low**
- **High**
- **Falling Edge**
- **Rising Edge**
- **Either Edge**

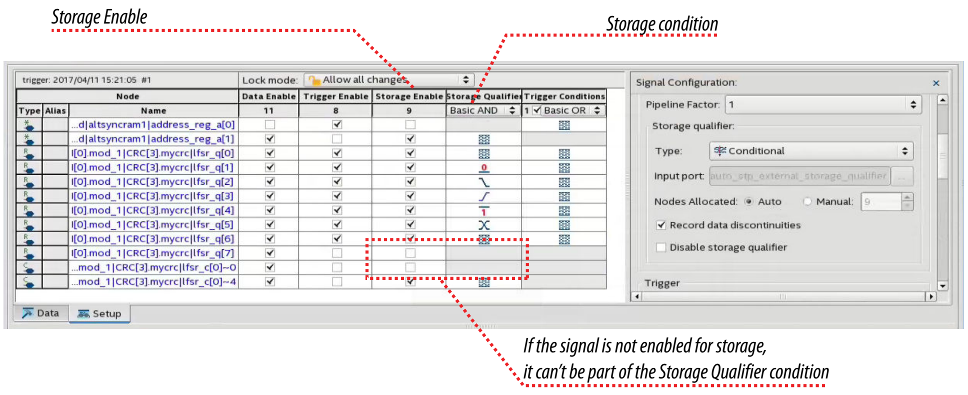
If you specify a **Basic AND** storage qualifier condition for more than one signal, the Signal Tap Logic Analyzer evaluates the logical AND of the conditions.

You can specify any other combinational or relational operators with the enabled signal set for storage qualification through advanced storage conditions.

You can define storage qualification conditions similar to the manner in which you define trigger conditions.

Figure 173. Conditional Storage Qualifier Setup

The figure details the conditional storage qualifier setup in the `.stp` file.


Storage Enable
Storage condition
If the signal is not enabled for storage,
it can't be part of the Storage Qualifier condition
Data
Setup

trigger: 2017/04/11 15:21:05 #1   Lock mode: Allow all changes

Node			Data Enable	Trigger Enable	Storage Enable	Storage Qualifier	Trigger Conditions
Type	Alias	Name	11	8	9	Basic AND	1 ✓ Basic OR
		...d|altsyncram1|address_reg_a[0]	☐	✓	☐		▨
		...d|altsyncram1|address_reg_a[1]	✓	☐	✓	▨	
		l[0].mod_1|CRC[3].mycrc|lfsr_q[0]	✓	✓	✓	▨	▨
		l[0].mod_1|CRC[3].mycrc|lfsr_q[1]	✓	✓	✓	0	▨
		l[0].mod_1|CRC[3].mycrc|lfsr_q[2]	✓	✓	✓	\	▨
		l[0].mod_1|CRC[3].mycrc|lfsr_q[3]	✓	✓	✓	/	▨
		l[0].mod_1|CRC[3].mycrc|lfsr_q[4]	✓	✓	✓	1	▨
		l[0].mod_1|CRC[3].mycrc|lfsr_q[5]	✓	✓	✓	X	▨
		l[0].mod_1|CRC[3].mycrc|lfsr_q[6]	✓	✓	✓	▨	▨
		l[0].mod_1|CRC[3].mycrc|lfsr_q[7]	✓	☐	☐		
		...mod_1|CRC[3].mycrc|lfsr_c[0]~0	✓	☐	☐		
		...mod_1|CRC[3].mycrc|lfsr_c[0]~4	✓	☐	✓	▨	

Signal Configuration:

Pipeline Factor: 1

Storage qualifier:

Type: Conditional

Input port: auto_stp_external_storage_qualifier

Nodes Allocated: [● Auto] ○ Manual: 9

✓ Record data discontinuities

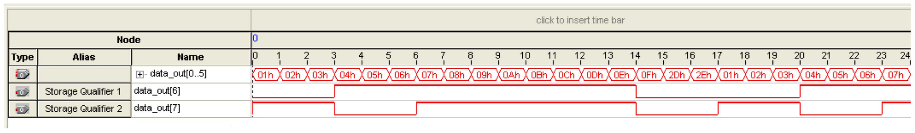
☐ Disable storage qualifier

Trigger

Figure 174. Comparing Continuous and Conditional Capture Mode in Data Acquisition of a Recurring Data Pattern

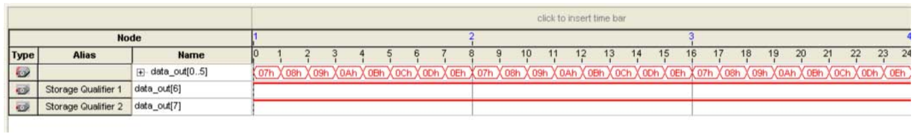
The data pattern is the same in both cases.

- Continuous sampling capture mode:



(1) Storage Qualifier condition is set up to evaluate data_out[6] AND data_out[7].

- Conditional sampling capture mode:



Related Links

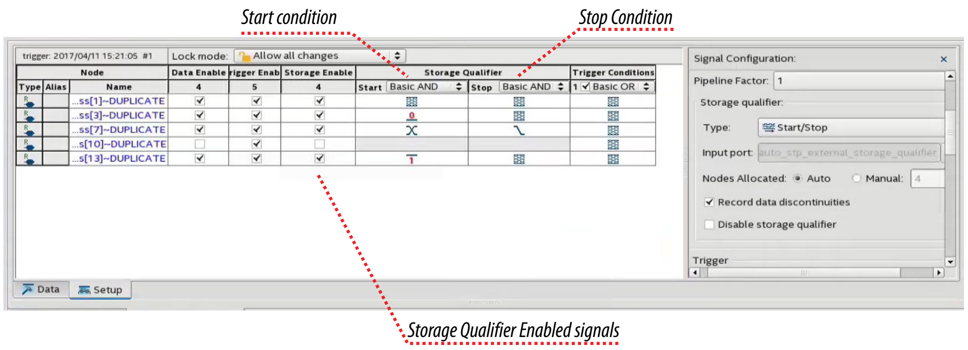
- Basic Trigger Conditions on page 352
- Comparison Trigger Conditions on page 353
- Advanced Trigger Conditions on page 355

14.3.9.4 Start/Stop Mode

The Start/Stop mode uses two sets of conditions, one to start data capture and one to stop data capture. If the start condition evaluates to TRUE, Signal Tap Logic Analyzer stores the buffer data every clock cycle until the stop condition evaluates to TRUE, which then pauses the data capture. The Logic Analyzer ignores additional start signals received after the data capture starts. If both start and stop evaluate to TRUE at the same time, the Logic Analyzer captures a single cycle.

Note: You can force a trigger by pressing the **Stop** button if the buffer fails to fill to completion due to a stop condition.

Figure 175. Start/Stop Mode Storage Qualifier Setup



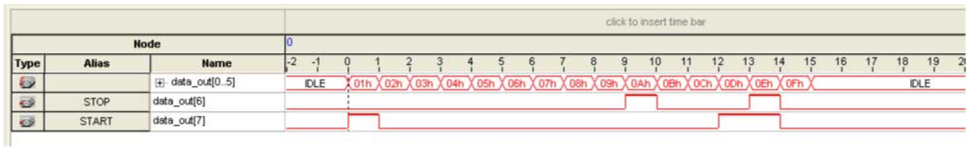
trigger: 2017/04/11 15:21:05 #1 Lock mode: Allow all changes

Node			Data Enable	rigger Enab	Storage Enable	Storage Qualifier				Trigger Conditions
Type	Alias	Name	4	5	4	Start	Basic AND	Stop	Basic AND	1 ✓ Basic OR
		...ss[1]~DUPLICATE	✓	✓	✓	▨		▨		▨
		...ss[3]~DUPLICATE	✓	✓	✓	0		▨		▨
		...ss[7]~DUPLICATE	✓	✓	✓	X		\		▨
		...s[10]~DUPLICATE		✓						▨
		...s[13]~DUPLICATE	✓	✓	✓	1		▨		▨

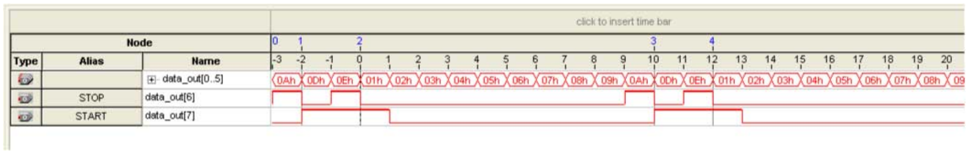
Signal Configuration:
Pipeline Factor: 1
Storage qualifier:
Type: Start/Stop
Input port: auto_stp_external_storage_qualifier
Nodes Allocated: [✓ Auto] Manual: 4
☑ Record data discontinuities
☐ Disable storage qualifier
Trigger

Figure 176. Comparing Continuous and Start/Stop Acquisition Modes for a Recurring Data Pattern

- Continuous Mode:



- Start/Stop Storage Qualifier:



14.3.9.5 State-Based

The State-based storage qualification mode is part of the State-based triggering flow. The state based triggering flow evaluates a conditional language to define how the Signal Tap Logic Analyzer writes data into the buffer. With the State-based trigger flow, you have command over boolean and relational operators to guide the execution flow for the target acquisition buffer.

When you enable the storage qualifier feature for the State-based flow, two additional commands become available: `start_store` and `stop_store`. These commands are similar to the Start/Stop capture conditions. Upon the start of acquisition, the Signal Tap Logic Analyzer doesn't write data into the buffer until a `start_store` action is performed. The `stop_store` command pauses the acquisition. If both `start_store` and `stop_store` actions occur within the same clock cycle, the Logic Analyzer stores a single sample into the acquisition buffer.

Related Links

State-Based Triggering on page 363

14.3.9.6 Showing Data Discontinuities

When you turn on **Record data discontinuities**, the Signal Tap Logic Analyzer marks the samples during which the acquisition paused from a storage qualifier. This marker is displayed in the waveform viewer after acquisition completes.

14.3.9.7 Disable Storage Qualifier

You can quickly turn off the storage qualifier with the **Disable Storage Qualifier** option, and perform a continuous capture. This option is run-time reconfigurable. Changing storage qualifier mode from the **Type** field requires a recompilation of the project.

Related Links

Runtime Reconfigurable Options on page 383

14.3.10 Manage Multiple Signal Tap Files and Configurations

You can debug different blocks in your design by grouping related monitoring signals. Likewise, you can use a group of signals to define multiple trigger conditions. Each combination of signals, capture settings, and trigger conditions determines a debug configuration, and one configuration can have zero or more associated data logs.

Signal Tap Logic Analyzer allows you to save debug configurations in more than one `.stp` file. Alternatively, you can embed multiple configurations within the same `.stp` file, and use the Data Log as a managing tool.

Note: Each `.stp` file is associated with a programming (`.sof`) file. To function correctly, the settings in the `.stp` file you use at runtime must match Signal Tap settings in the `.sof` file you use to program the device.

Related Links

Ensure Setting Compatibility Between .stp and .sof Files on page 382

14.3.10.1 Data Log Pane

The Data Log pane displays all Signal Tap configurations and data capture results stored within a single `.stp` file.

- To save the current configuration or capture in the Data Log—and `.stp` file, click **Edit ➤ Save to Data Log**. Alternatively, click the **Save to Data Log** icon at the top of the Data Log pane.
- To generate a log entry after every data capture, click **Edit ➤ Enable Data Log**. Alternatively, check the box at the top of the Data Log pane.

The Data Log displays its contents in a tree hierarchy. The active items display a different icon.

Table 110. Data Log Items

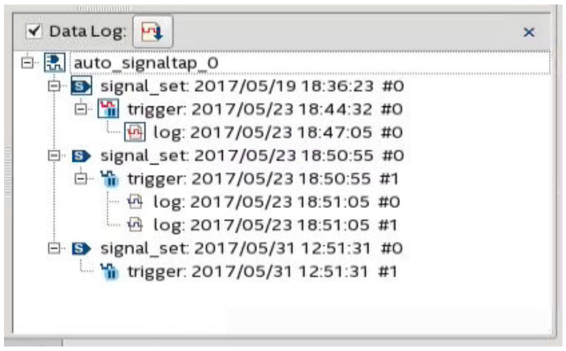
Item	Icon		Contains one or more	Comments
	Unselected	Selected		
Instance			Signal Set	
Signal Set			Trigger	The Signal Set changes whenever you add a new signal to Signal Tap. After a change in the Signal Set, you need to recompile.
Trigger			Capture Log	A trigger changes when you change any trigger condition. These changes do not require recompilation.
Capture Log				

The name on each entry displays the wall-clock time when Signal Tap Logic Analyzer triggered, and the time elapsed from start acquisition to trigger activation. You can rename entries so they make sense to you.

To switch between configurations, double-click an entry in the Data Log. As a result, the **Setup** tab updates to display the active signal list and trigger conditions.

Example 34. Simple Data Log

On this example, the Data Log displays one instance with three signal set configurations.



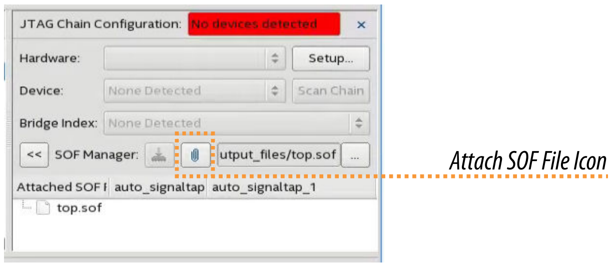
```
Data Log:
auto_signaltap_0
  signal_set: 2017/05/19 18:36:23 #0
    trigger: 2017/05/23 18:44:32 #0
      log: 2017/05/23 18:47:05 #0
  signal_set: 2017/05/23 18:50:55 #0
    trigger: 2017/05/23 18:50:55 #1
      log: 2017/05/23 18:51:05 #0
      log: 2017/05/23 18:51:05 #1
  signal_set: 2017/05/31 12:51:31 #0
    trigger: 2017/05/31 12:51:31 #1
```

14.3.10.2 SOF Manager

The SOF Manager is in the **JTAG Chain Configuration** pane.

With the SOF Manager you can embed multiple SOFs into one `.stp` file. This action lets you move the `.stp` file to a different location, either on the same computer or across a network, without including the associated `.sof` separately. To embed a new SOF in the `.stp` file, click the **Attach SOF File** icon .

Figure 177. SOF Manager



JTAG Chain Configuration: No devices detected

Hardware: | Setup...

Device: None Detected | Scan Chain

Bridge Index: None Detected

SOF Manager: utput_files/top.sof

Attached SOF I | auto_signaltap | auto_signaltap_1

top.sof

As you switch between configurations in the Data Log, you can extract the SOF that is compatible with that configuration.

To download the new SOF to the FPGA, click the Program Device icon in the SOF Manager, after ensuring that the configuration of your `.stp` matches the design programmed into the target device.

Related Links

Data Log Pane on page 350

14.4 Defining Triggers

You specify various types of trigger conditions using the Signal Tap Logic Analyzer on the **Signal Configuration** pane. When you start the Signal Tap Logic Analyzer, it samples activity continuously from the monitored signals. The Signal Tap Logic Analyzer “triggers”—that is, the logic analyzer stops and displays the data—when a condition or set of conditions that you specified have been reached.

14.4.1 Basic Trigger Conditions

If you select the **Basic AND** or **Basic OR** trigger type, you must specify the trigger pattern for each signal you added in the `.stp`. To specify the trigger pattern, right-click the **Trigger Conditions** column and click the desired pattern. Set the trigger pattern to any of the following conditions:

- **Don’t Care**
- **Low**
- **High**
- **Falling Edge**
- **Rising Edge**
- **Either Edge**

For buses, type a pattern in binary, or right-click and select **Insert Value** to enter the pattern in other number formats. Note that you can enter X to specify a set of “don’t care” values in either your hexadecimal or your binary string. For signals in the `.stp` file that have an associated mnemonic table, you can right-click and select an entry from the table to specify pre-defined conditions for the trigger.

When you add signals through plug-ins, you can create basic triggers using predefined mnemonic table entries. For example, with the Nios II plug-in, if you specify an `.elf` file from your Nios II IDE design, you can type the name of a function from your Nios II code. The logic analyzer triggers when the Nios II instruction address matches the address of the code function name that you specify.

Data capture stops and the Logic Analyzer stores the data in the buffer when the logical AND of all the signals for a given trigger condition evaluates to TRUE.

Related Links

View, Analyze, and Use Captured Data on page 386

14.4.1.1 Using the Basic OR Trigger Condition with Nested Groups

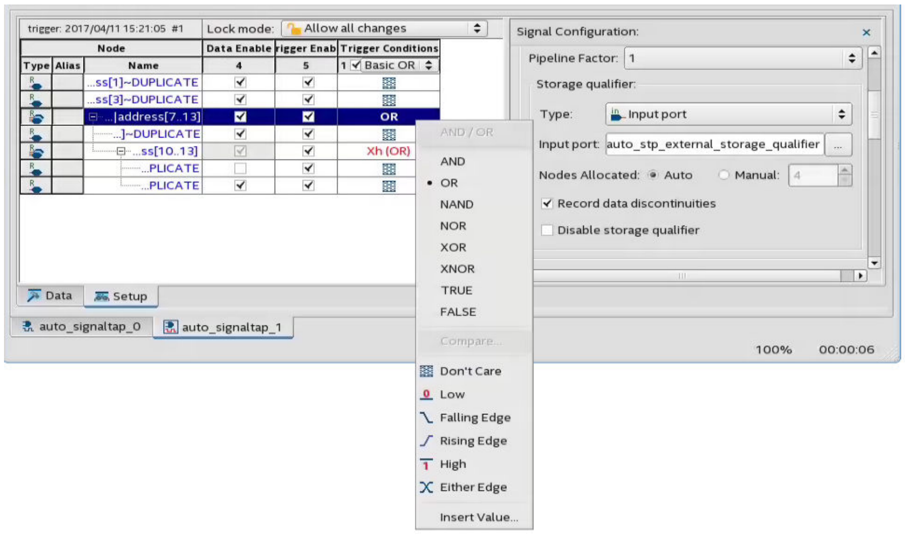
When you specify a set of signals as a nested group (group of groups) with the **Basic OR** trigger type, Signal Tap Logic Analyzer generates an advanced trigger condition. This condition sorts signals within groups to minimize the need to recompile your design. As long as the parent-child relationships of nodes are kept constant, the advanced trigger condition does not change. You can modify the sibling relationships of nodes and not need to recompile your design.

The evaluation precedence of a nested trigger condition starts at the bottom-level with the leaf-groups. The Logic Analyzer uses the resulting logic value to compute the parent group's logic value. If you manually set the value of a group, the logic value of the group's members doesn't influence the result of the group trigger. To create a nested trigger condition:

1. Select **Basic OR** under **Trigger Conditions**.
2. In the **Setup** tab, select several nodes. Include groups in your selection.
3. Right-click the **Setup** tab and select **Group**.
4. Select the nested group and right-click to set a group trigger condition that applies the reduction **AND**, **OR**, **NAND**, **NOR**, **XOR**, **XNOR**, or logical **TRUE** or **FALSE**.

   Note: You can only select OR and AND group trigger conditions for bottom-level groups (groups with no groups as children).

Figure 178. Applying Trigger Condition to Nested Group



trigger: 2017/04/11 15:21:05 #1 Lock mode: Allow all changes

Type	Alias	Name	Data Enable 4	Trigger Enable 5	Trigger Conditions 1 ✓ Basic OR
		...ss[1]~DUPLICATE	✓	✓	▨
		...ss[3]~DUPLICATE	✓	✓	▨
		⊟ ...|address[7..13]	✓	✓	OR
		...]~DUPLICATE	✓	✓	▨
		⊟ ...ss[10..13]	✓	✓	Xh (OR)
		...PLICATE		✓	▨
		...PLICATE	✓	✓	▨

Signal Configuration:
Pipeline Factor: 1
Storage qualifier:
Type: Input port
Input port: auto_stp_external_storage_qualifier
Nodes Allocated: ● Auto ○ Manual: 4
✓ Record data discontinuities
☐ Disable storage qualifier

100% 00:00:06

14.4.2 Comparison Trigger Conditions

The **Comparison** trigger allows you to compare multiple grouped bits of a bus to an expected integer value by specifying simple comparison conditions on the bus node. The **Comparison** trigger preserves all the trigger conditions that the **Basic OR** trigger includes. You can use the **Comparison** trigger in combination with other triggers. You can also switch between **Basic OR** trigger and **Comparison** trigger at run-time, without the need for recompilation.

Signal Tap Logic Analyzer supports the following types of **Comparison** trigger conditions:

- **Single-value comparison**—compares a bus node's value to a numeric value that you specify. Use one of these operands for comparison: >, >=, ==, <=, <. Returns 1 when the bus node matches the specified numeric value.
- **Interval check**—verifies whether a bus node's value confines to an interval that you define. Returns 1 when the bus node's value lies within the specified bounded interval.

Follow these rules when using the **Comparison** trigger condition:

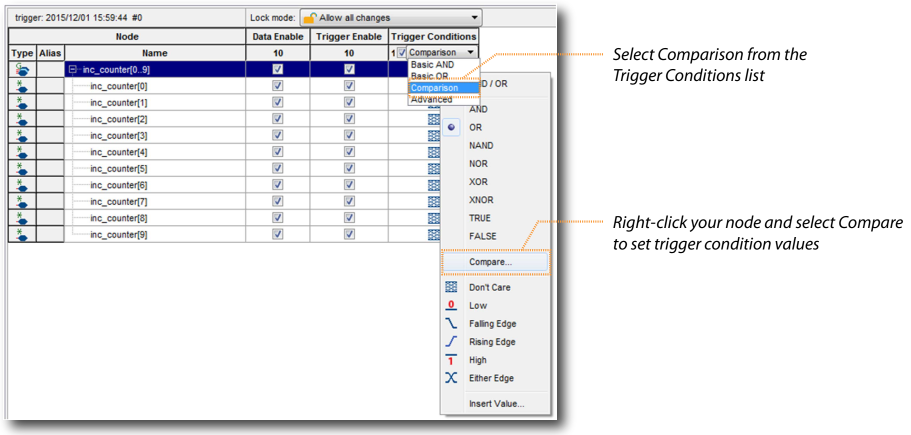
- Apply the **Comparison** trigger only to bus nodes consisting of leaf nodes.
- Do not form sub-groups within a bus node.
- Do not enable or disable individual trigger nodes within a bus node.
- Do not specify comparison values (in case of single-value comparison) or boundary values (in case of interval check) exceeding the selected node's bus-width.

14.4.2.1 Specifying the Comparison Trigger Conditions

Follow these steps to specify the **Comparison** trigger conditions:

1. From the **Setup** tab, select **Comparison** under **Trigger Conditions**.
2. Right-click the node in the trigger editor, and select **Compare**.

Figure 179. Selecting the Comparison Trigger Condition



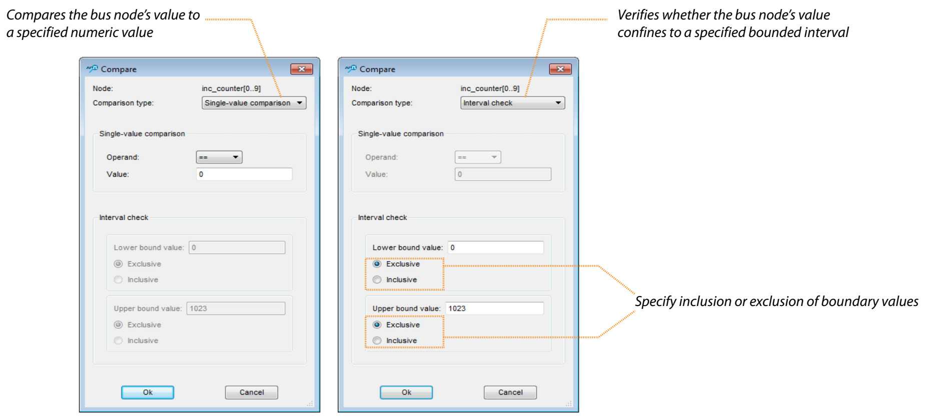
trigger: 2015/12/01 15:59:44 #0 Lock mode: Allow all changes

Node			Data Enable	Trigger Enable	Trigger Conditions
Type	Alias	Name	10	10	1 ✓ Comparison
		inc_counter[0..9]	✓	✓	
		inc_counter[0]	✓	✓	
		inc_counter[1]	✓	✓	
		inc_counter[2]	✓	✓	▨
		inc_counter[3]	✓	✓	▨
		inc_counter[4]	✓	✓	▨
		inc_counter[5]	✓	✓	▨
		inc_counter[6]	✓	✓	▨
		inc_counter[7]	✓	✓	▨
		inc_counter[8]	✓	✓	▨
		inc_counter[9]	✓	✓	▨

3. Select the **Comparison type** from the Compare window.
   — If you choose **Single-value comparison** as your comparison type, specify the operand and value.
   — If you choose **Interval check** as your comparison type, provide the lower and upper bound values for the interval.

   You can also specify if you want to include or exclude the boundary values.

Figure 180. Specifying the Comparison Values


Compares the bus node's value to a specified numeric value
Verifies whether the bus node's value confines to a specified bounded interval
Specify inclusion or exclusion of boundary values
Compare
Ok
Cancel

Compare

Node: inc_counter[0..9]
Comparison type: Single-value comparison
Single-value comparison
Operand: ==
Value: 0
Interval check
Lower bound value: 0
(●) Exclusive ( ) Inclusive
Upper bound value: 1023
(●) Exclusive ( ) Inclusive

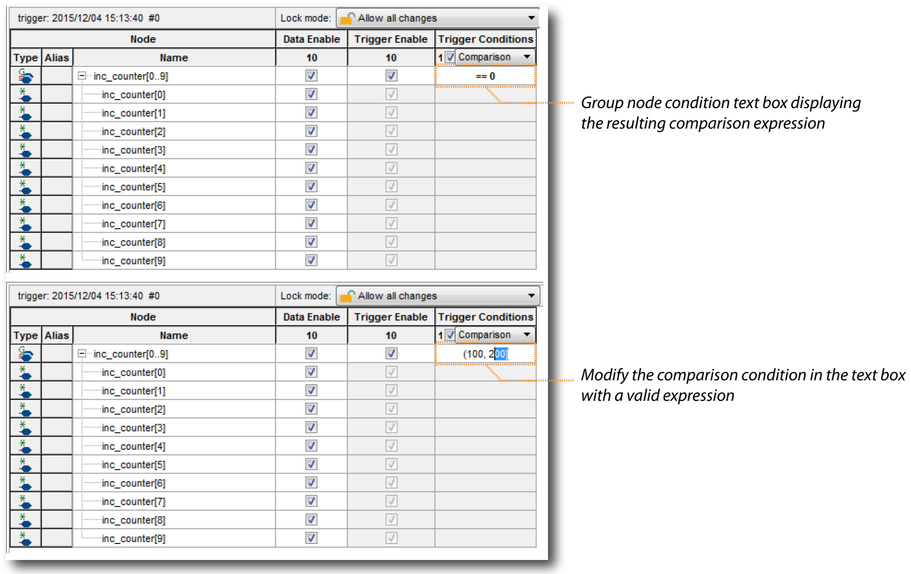
Compare

Node: inc_counter[0..9]
Comparison type: Interval check
Single-value comparison
Operand: ==
Value: 0
Interval check
Lower bound value: 0
(●) Exclusive ( ) Inclusive
Upper bound value: 1023
(●) Exclusive ( ) Inclusive

4. Click **OK**. The trigger editor displays the resulting comparison expression in the group node condition text box.

   Note: You can modify the comparison condition in the text box with a valid expression.

Figure 181. Resulting Comparison Condition in Text Box


Group node condition text box displaying the resulting comparison expression
Modify the comparison condition in the text box with a valid expression

trigger: 2015/12/04 15:13:40 #0 Lock mode: Allow all changes

Node			Data Enable	Trigger Enable	Trigger Conditions
Type	Alias	Name	10	10	1 ✓ Comparison
		inc_counter[0..9]	✓	✓	== 0
		inc_counter[0]	✓	✓	
		inc_counter[1]	✓	✓	
		inc_counter[2]	✓	✓	
		inc_counter[3]	✓	✓	
		inc_counter[4]	✓	✓	
		inc_counter[5]	✓	✓	
		inc_counter[6]	✓	✓	
		inc_counter[7]	✓	✓	
		inc_counter[8]	✓	✓	
		inc_counter[9]	✓	✓	

trigger: 2015/12/04 15:13:40 #0 Lock mode: Allow all changes

Node			Data Enable	Trigger Enable	Trigger Conditions
Type	Alias	Name	10	10	1 ✓ Comparison
		inc_counter[0..9]	✓	✓	(100, 200)
		inc_counter[0]	✓	✓	
		inc_counter[1]	✓	✓	
		inc_counter[2]	✓	✓	
		inc_counter[3]	✓	✓	
		inc_counter[4]	✓	✓	
		inc_counter[5]	✓	✓	
		inc_counter[6]	✓	✓	
		inc_counter[7]	✓	✓	
		inc_counter[8]	✓	✓	
		inc_counter[9]	✓	✓	

14.4.3 Advanced Trigger Conditions

To capture data for a given combination of conditions, build an advanced trigger. The Signal Tap Logic Analyzer provides the **Advanced Trigger** tab, which helps you build a complex trigger expression using a GUI.

Open the **Advanced Trigger** tab by selecting **Advanced** in the **Trigger Conditions** drop-down menu.

Figure 182. Accessing the Advanced Trigger Condition Tab

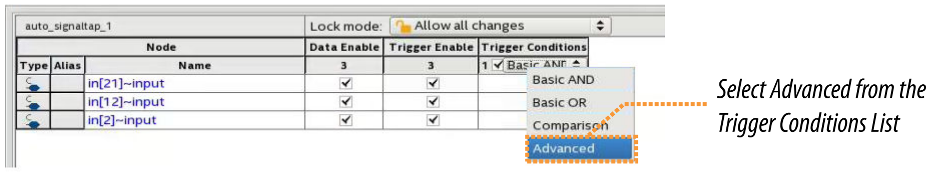
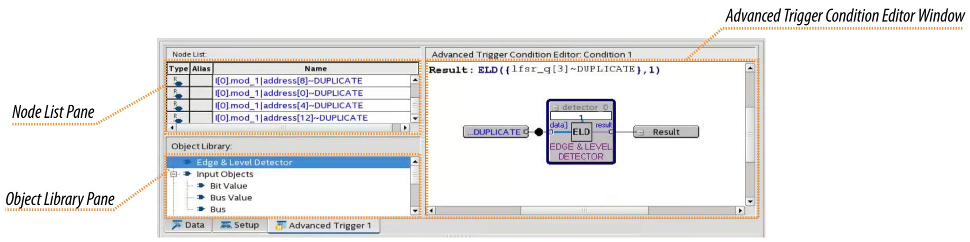


Figure 183. Advanced Trigger Condition Tab


Advanced Trigger Condition Editor Window
Node List Pane
Object Library Pane
Node List:
Type
Alias
Name
l[0].mod_1|address[8]~DUPLICATE
l[0].mod_1|address[0]~DUPLICATE
l[0].mod_1|address[4]~DUPLICATE
l[0].mod_1|address[12]~DUPLICATE
Object Library:
Edge & Level Detector
Input Objects
Bit Value
Bus Value
Bus
Data
Setup
Advanced Trigger 1
Advanced Trigger Condition Editor: Condition 1
Result: ELD({lfsr_q[3]~DUPLICATE},1)
DUPLICATE
detector
data
ELD
result
EDGE & LEVEL DETECTOR
Result

To build a complex trigger condition in an expression tree, drag-and-drop operators from the **Object Library** pane and the **Node List** pane into the **Advanced Trigger Configuration Editor** window.

To configure the operators' settings, double-click or right-click the operators that you placed and click **Properties**.

Table 111. Advanced Triggering Operators

Category	Name
Signal Detection	Edge and Level Detector
Input Objects	Bit Bit Value Bus Bus Value
Comparison	Less Than Less Than or Equal To Equality Inequality Greater Than or Equal To Greater Than
Bitwise	Bitwise Complement Bitwise AND Bitwise OR Bitwise XOR
Logical	Logical NOT Logical AND Logical OR Logical XOR
Reduction	Reduction AND Reduction OR Reduction XOR
Shift	Left Shift

continued...

Category	Name
	Right Shift
Custom Trigger HDL	

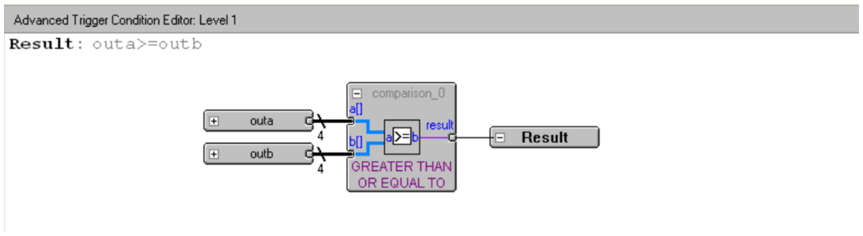
Adding many objects to the Advanced Trigger Condition Editor can make the work space cluttered and difficult to read. To keep objects organized while you build your advanced trigger condition, use the shortcut menu and select **Arrange All Objects**. Alternatively, use the **Zoom-Out** command to fit more objects into the **Advanced Trigger Condition Editor** window.

14.4.3.1 Examples of Advanced Triggering Expressions

The following examples show how to use Advanced Triggering:

Figure 184. Bus outa Is Greater Than or Equal to Bus outb

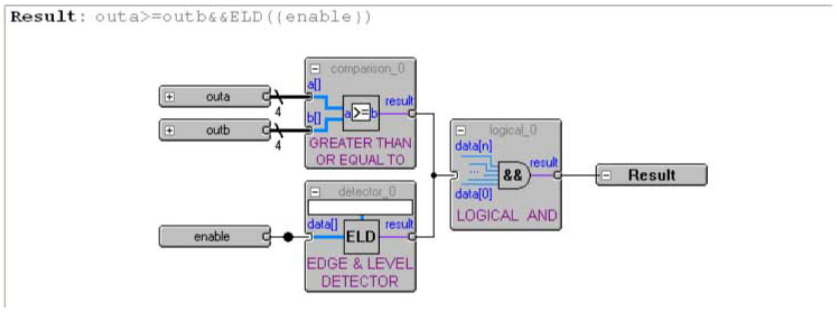
Trigger when bus outa is greater than or equal to outb.



Result: outa>=outb

Figure 185. Enable Signal Has a Rising Edge

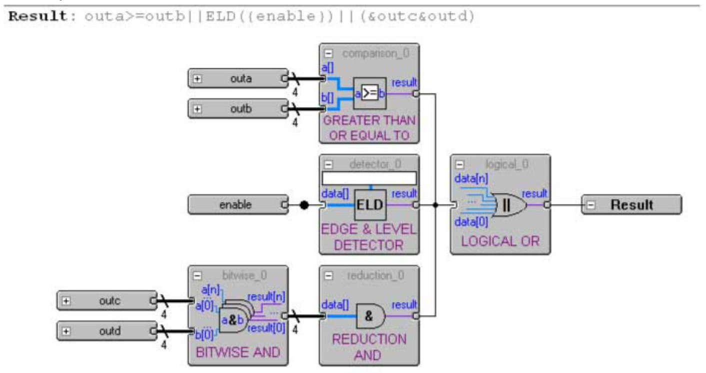
Trigger when bus outa is greater than or equal to bus outb, and when the enable signal has a rising edge.



Result: outa>=outb&&ELD((enable))

Figure 186. Bitwise AND Operation

Trigger when bus `outa` is greater than or equal to bus `outb`, or when the enable signal has a rising edge. Or, when a bitwise AND operation has been performed between bus `outc` and bus `outd`, and all bits of the result of that operation are equal to 1.

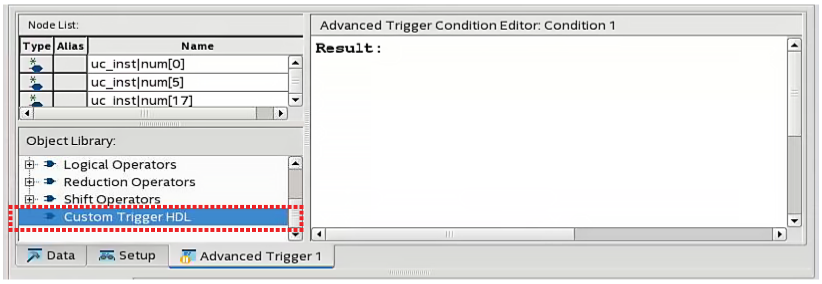

```
Result: outa>=outb||ELD((enable))||(&outc&outd)
```

14.4.4 Custom Trigger HDL Object

Signal Tap Logic Analyzer allows you to use your own HDL module to create a custom trigger condition. You can use the Custom Trigger HDL object to simulate your triggering logic and ensure that the logic itself is not faulty. Additionally, you can tap instances of modules anywhere in the hierarchy of your design, without having to manually route all the necessary connections.

The Custom Trigger HDL object appears in the **Object Library** pane of the **Advanced Trigger** editor.

Figure 187. Object Library

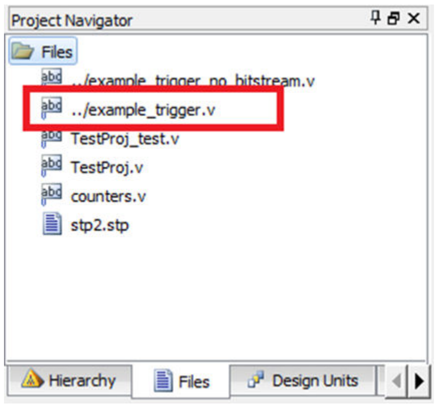


14.4.4.1 Using the Custom Trigger HDL Object

To define a custom trigger flow:

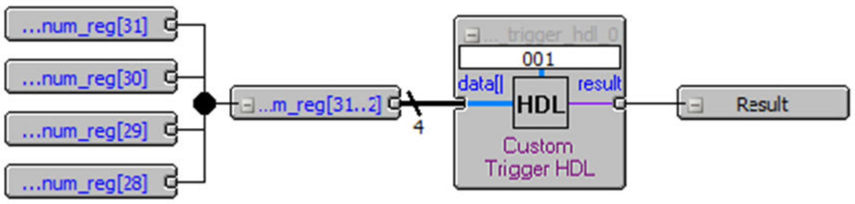
1. Select the trigger you want to edit.
2. Open the **Advanced Trigger** tab by selecting **Advanced** in the **Trigger Conditions** drop-down menu.
3. Add to your project the HDL source file that contains the trigger module using the **Project Navigator**.
   — Alternatively, append the HDL for your trigger module to a source file already included in the project.

Figure 188. HDL Trigger in the Project Navigator



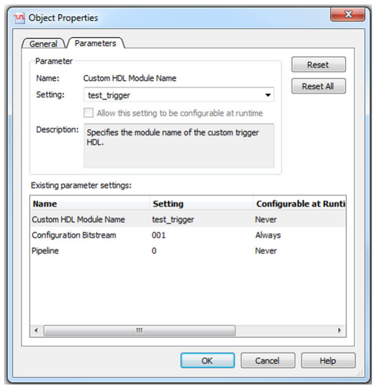
4. Implement the inputs and outputs that your Custom Trigger HDL module requires.
5. Drag in your Custom Trigger HDL object and connect the object's data input bus and result output bit to the final trigger result.

Figure 189. Custom Trigger HDL Object



6. Right-click your Custom Trigger HDL object and configure the object's properties.

Figure 190. Configure Object Properties



Parameter

Name: Custom HDL Module Name

Setting: test_trigger

Allow this setting to be configurable at runtime

Description: Specifies the module name of the custom trigger HDL.

Existing parameter settings:

Name	Setting	Configurable at Runti
Custom HDL Module Name	test_trigger	Never
Configuration Bitstream	001	Always
Pipeline	0	Never

7. Compile your design.
8. Acquire data with Signal Tap using your custom Trigger HDL object.

Example 35. Verilog HDL Triggers

The following trigger uses configuration bitstream:

```
module test_trigger
    (
        input acq_clk, reset,
        input[3:0] data_in,
        input[1:0] pattern_in,
        output reg trigger_out
    );
    always @(pattern_in) begin
        case (pattern_in)
            2'b00:
                trigger_out = &data_in;
            2'b01:
                trigger_out = |data_in;
            2'b10:
                trigger_out = 1'b0;
            2'b11:
                trigger_out = 1'b1;
        endcase
    end
endmodule
```

This trigger does not have configuration bitstream:

```
module test_trigger_no_bs
    (
        input acq_clk, reset,
        input[3:0] data_in,
        output reg trigger_out
    );
    assign trigger_out = &data_in;
endmodule
```

14.4.4.2 Required Inputs and Outputs of Custom Trigger HDL Module

Table 112. Custom Trigger HDL Module Required Inputs and Outputs

Name	Description	Input/Output	Required/ Optional
`acq_clk`	Acquisition clock that Signal Tap uses	Input	Required
`reset`	Reset that Signal Tap uses when restarting a capture.	Input	Required
`data_in`	• Data input you connect in the Advanced Trigger editor. • Data your module uses to trigger.	Input	Required
`pattern_in`	• Module's input for the configuration bitstream property. • Runtime configurable property that you can set from Signal Tap GUI to change the behavior of your trigger logic.	Input	Optional
`trigger_out`	Output signal of your module that asserts when trigger conditions met.	Output	Required

14.4.4.3 Properties of Custom Trigger HDL Module

Table 113. Custom Trigger HDL Module Properties

Property	Description
Custom HDL Module Name	Module name of your triggering logic.
Configuration Bitstream	• Allows you to create runtime-configurable trigger logic which can change its behavior based upon the value of the configuration bitstream. • The configuration bitstream property is read as binary, therefore it must contain only the characters `1` and `0`. The bit-width (number of `1`s and `0`s) must match the `pattern_in` bit width. • A blank configuration bitstream implies that your module does not have a `pattern_in` input.
Pipeline	Specifies the number of pipeline stages in your triggering logic. For example, if after receiving a triggering input the LA needs three clock cycles to assert the trigger output, you can denote a pipeline value of three.

14.4.5 Trigger Condition Flow Control

The Trigger Condition Flow allows you to define the relationship between a set of triggering conditions. Signal Tap Logic Analyzer **Signal Configuration** pane offers two flow control mechanisms for organizing trigger conditions:

- **Sequential Triggering**—default triggering flow. Sequential triggering allows you to define up to 10 triggering levels that must be satisfied before the acquisition buffer finishes capturing.
- **State-Based Triggering**—gives the greatest control over your acquisition buffer. Custom-based triggering allows you to organize trigger conditions into states based on a conditional flow that you define.

You can use sequential or state based triggering with either a segmented or a non-segmented buffer.

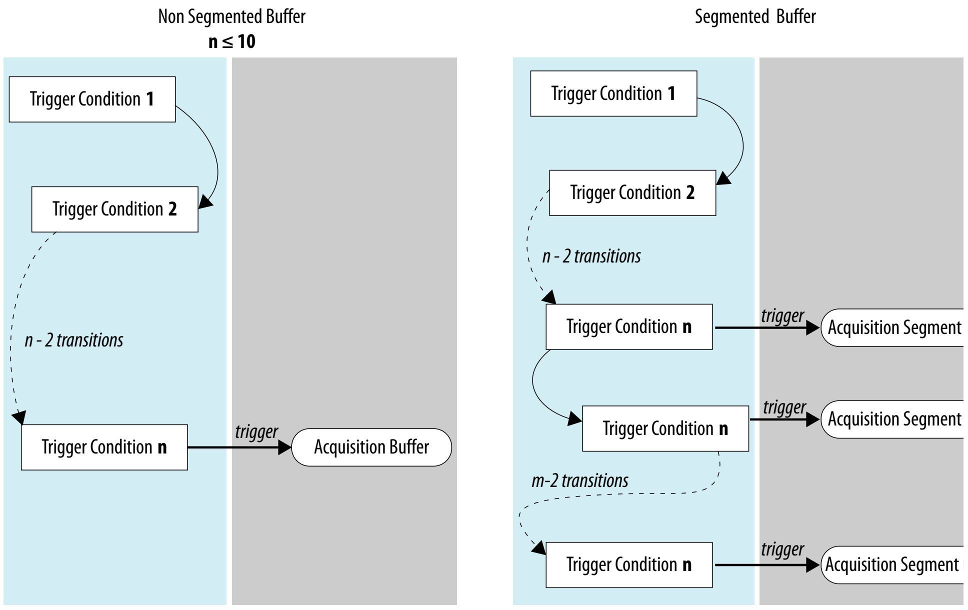
14.4.5.1 Sequential Triggering

Sequential triggering flow allows you to cascade up to 10 levels of triggering conditions. Signal Tap Logic Analyzer sequentially evaluates each of the conditions.

When the last triggering condition evaluates to TRUE, the Signal Tap Logic Analyzer triggers the acquisition buffer. For segmented buffers, every acquisition segment after the first triggers on the last condition that you specified. Use the Simple Sequential Triggering feature with basic triggers, comparison triggers, advanced triggers, or a mix of all three. The figure illustrates the simple sequential triggering flow for non-segmented and segmented buffers.

The external trigger is considered as trigger level `0`. The external trigger must be evaluated before the main trigger levels are evaluated.

Figure 191. Sequential Triggering Flow



Notes to figure:

1. The acquisition buffer starts capture when all n triggering levels are satisfied, where $n \leq 10$.
2. If you define an external trigger input, the Logic Analyzer evaluates it before evaluating all other trigger conditions.

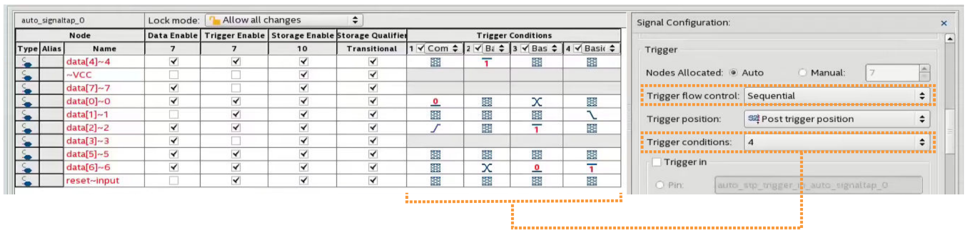
14.4.5.1.1 Configuring the Sequential Triggering Flow

To configure Signal Tap Logic Analyzer for sequential triggering:

1. On **Trigger Flow Control**, select **Sequential**
2. On **Trigger Conditions**, select the number of trigger conditions from the drop-down list.
   The **Node List** pane now displays the same number of trigger condition columns.
3. Configure each trigger condition in the **Node List** pane.

   You can enable/disable any trigger condition from the column header.

Figure 192. Sequential Triggering Flow Configuration



auto_signaltap_0 Lock mode: Allow all changes

Node			Data Enable	Trigger Enable	Storage Enable	Storage Qualifier	Trigger Conditions			
Type	Alias	Name	7	7	10	Transitional	1 ✓ Com	2 ✓ Ba	3 ✓ Bas	4 ✓ Basic
		data[4]~4	✓	✓	✓	✓	▨	1	▨	▨
		~VCC			✓	✓				
		data[7]~7	✓		✓	✓				
		data[0]~0	✓	✓	✓	✓	0	▨	X	▨
		data[1]~1		✓	✓	✓	▨	▨	▨	╲
		data[2]~2	✓	✓	✓	✓	╱	▨	1	▨
		data[3]~3	✓		✓	✓				
		data[5]~5	✓	✓	✓	✓	▨	▨	▨	▨
		data[6]~6	✓	✓	✓	✓	▨	X	0	1
		reset~input		✓	✓	✓	▨	▨	▨	▨

Signal Configuration:

Trigger

Nodes Allocated: [● Auto] ○ Manual: 7

Trigger flow control: Sequential

Trigger position: Post trigger position

Trigger conditions: 4

☐ Trigger in

○ Pin: auto_stp_trigger_in_auto_signaltap_0

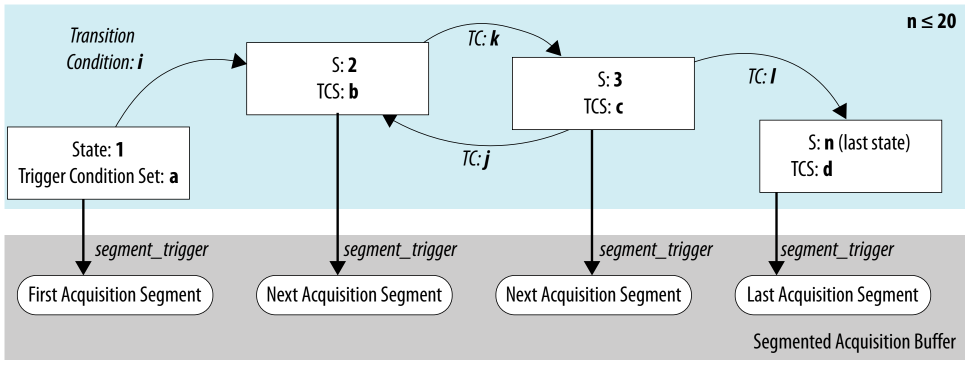
14.4.5.2 State-Based Triggering

With state-based triggering, a state diagram organizes the events that trigger the acquisition buffer. The states capture all actions that the acquisition buffer performs, and each state contains conditional expressions that define transition conditions.

Custom state-based triggering grants control over triggering condition arrangement, and allows for more efficient use of the space available in the acquisition buffer, because the Logic Analyzer only captures samples of interest.

To help you describe the relationship between triggering conditions, the state-based triggering flow provides tooltips within the flow GUI. Additionally, you can use the Signal Tap Trigger Flow Description Language, which is based upon conditional expressions.

Figure 193. State-Based Triggering Flow



Notes to figure:

1. You can define up to 20 different states.
2. If you define an external trigger input, the logic analyzer evaluates it before any conditions in the custom state-based triggering flow.

Each state allows you to define a set of conditional expressions. Each conditional expression is a Boolean expression that depends on a combination of triggering conditions, counters, and status flags. You configure the triggering conditions within the **Setup** tab. The Signal Tap Logic Analyzer custom-based triggering flow provides counters and status flags.

Within each conditional expression you define a set of actions. Actions include triggering the acquisition buffer to stop capture, a modification to either a counter or status flag, or a state transition.

Trigger actions can apply to either a single segment of a segmented acquisition buffer or to the entire non-segmented acquisition buffer. Each trigger action provides you with an optional count that specifies the number of samples the buffer captures before the logic analyzer stops acquisition of the current segment. The count argument allows you to control the amount of data the buffer captures before and after a triggering event occurs.

Resource manipulation actions allow you to increment and decrement counters or set and clear status flags. The logic analyzer uses counter and status flag resources as optional inputs in conditional expressions. Counters and status flags are useful for counting the number of occurrences of certain events and for aiding in triggering flow control.

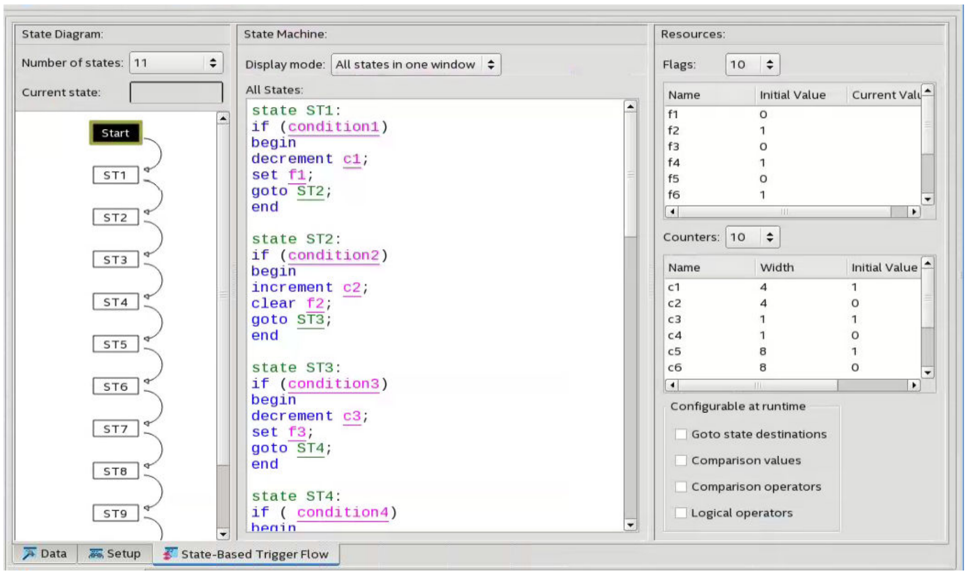
The state-based triggering flow allows you to capture a sequence of events that may not necessarily be contiguous in time. For example, a communication transaction between two devices that includes a hand shaking protocol containing a sequence of acknowledgements.

14.4.5.2.1 State-Based Triggering Flow Tab

The **State-Based Trigger Flow** tab is the control interface for the custom state-based triggering flow.

This tab is only available when you select **State-Based** on the **Trigger Flow Control** list. If you specify **Trigger Flow Control** as **Sequential**, the **State-Based Trigger Flow** tab is not visible.

Figure 194. State-Based Triggering Flow Tab



Number of states: 11

Display mode: All states in one window

```
state ST1:
if (condition1)
begin
decrement c1;
set f1;
goto ST2;
end

state ST2:
if (condition2)
begin
increment c2;
clear f2;
goto ST3;
end

state ST3:
if (condition3)
begin
decrement c3;
set f3;
goto ST4;
end

state ST4:
if ( condition4)
begin
```

Flags: 10

Name	Initial Value	Current Valu
f1	0	
f2	1	
f3	0	
f4	1	
f5	0	
f6	1	

Counters: 10

Name	Width	Initial Value
c1	4	1
c2	4	0
c3	1	1
c4	1	0
c5	8	1
c6	8	0

- [ ] Goto state destinations
- [ ] Comparison values
- [ ] Comparison operators
- [ ] Logical operators

The **State-Based Trigger Flow** tab contains three panes:

State Diagram Pane

The **State Diagram** pane provides a graphical overview of your triggering flow. this pane displays the number of available states and the state transitions. To adjust the number of available states, use the menu above the graphical overview.

State Machine Pane

The **State Machine** pane contains the text entry boxes where you define the triggering flow and actions associated with each state.

- You can define the triggering flow using the Signal Tap Trigger Flow Description Language, a simple language based on "if-else" conditional statements.
- Tooltips appear when you move the mouse over the cursor, to guide command entry into the state boxes.
- The GUI provides a syntax check on your flow description in real-time and highlights any errors in the text flow.

The State Machine description text boxes default to show one text box per state. You can also have the entire flow description shown in a single text field. This option can be useful when copying and pasting a flow description from a template or an external text editor. To toggle between one window per state, or all states in one window, select the appropriate option under **State Display mode.**

Related Links

Signal Tap Trigger Flow Description Language on page 366

Resources Pane

The **Resources** pane allows you to declare status flags and counters for your Custom Triggering Flow's conditional expressions.

- You can increment/decrement counters or set/clear status flags within your triggering flow.
- You can specify up to 20 counters and 20 status flags.
- To initialize counter and status flags, right-click the row in the table and select **Set Initial Value.**
- To specify a counter width, right-click the counter in the table and select **Set Width**.
- To assist in debugging your trigger flow specification, the logic analyzer dynamically updates counters and flag values after acquisition starts.

The **Configurable at runtime** settings allow you to control which options can change at runtime without requiring a recompilation.

Table 114. Runtime Reconfigurable Settings, State-Based Triggering Flow

Setting	Description
Destination of `goto` action	Allows you to modify the destination of the state transition at runtime.
Comparison values	Allows you to modify comparison values in Boolean expressions at runtime. In addition, you can modify the `segment_trigger` and trigger action post-fill count argument at runtime.
Comparison operators	Allows you to modify the operators in Boolean expressions at runtime.
Logical operators	Allows you to modify the logical operators in Boolean expressions at runtime.

Related Links

- Performance and Resource Considerations on page 380
- Runtime Reconfigurable Options on page 383

14.4.5.2.2 Trigger Lock Mode

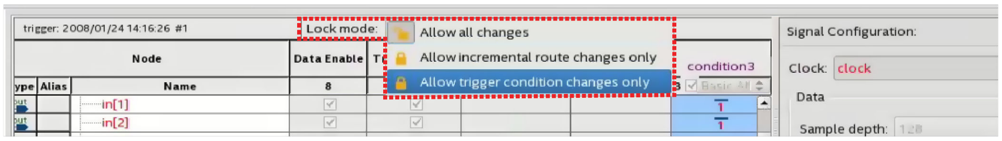
Trigger lock mode restricts changes to only the configuration settings that you specify as **Configurable at runtime**. The runtime configurable settings for the Custom Trigger Flow tab are on by default.

Note: You may get some performance advantages by disabling some of the runtime configurable options.

You can restrict changes to your Signal Tap configuration to include only the options that do not require a recompilation. Trigger lock-mode allows you to make changes that reflect immediately in the device.

1. On the **Setup** tab, point to **Lock Mode** and select **Allow trigger condition changes only**.

Figure 195. Allow Trigger Conditions Change Only



2. Modify the Trigger Flow conditions.

Incremental Route lock-mode restricts the GUI to only allow changes that require an Incremental Route compilation using Rapid Recompile. Use Rapid Recompile to perform incremental routing and gain a 2-4x speedup over the initial full compilation.

14.4.5.3 Signal Tap Trigger Flow Description Language

The Trigger Flow Description Language is based on a list of conditional expressions per state to define a set of actions.

To describe the actions the Logic Analyzer evaluates when a state is reached, you follow this syntax:

Syntax of Trigger Flow Description Language

```
state <state_label>:
    <action_list>
    if (<boolean_expression>)
        <action_list>
    [else if (<boolean_expression>)
        <action_list>]
    [else
        <action_list>]
```

- Non-terminals are delimited by "<>".
- Optional arguments are delimited by "[]"
- The priority for evaluation of conditional statements is from top to bottom.
- The Trigger Flow Description Language allows multiple `else if` conditions.

<state_label> on page 367

<boolean_expression> on page 367

<action_list> on page 368

Related Links

Custom Triggering Flow Application Examples on page 397

14.4.5.3.1 <state_label>

Identifies a given state. You use the state label to start describing the actions the Logic Analyzer evaluates once said state is reached. You can also use the state label with the `goto` command.

The state description header syntax is:
`state` <*state_label*>

The description of a state ends with the beginning of another state or the end of the whole trigger flow description.

14.4.5.3.2 <boolean_expression>

Collection of operators and operands that evaluate into a Boolean result. The operators can be logical or relational. Depending on the operator, the operand can reference a trigger condition, a counter and a register, or a numeric value. To group a set of operands within an expression, you use parentheses.

Table 115. Logical Operators

Logical operators accept any boolean expression as an operand.

Operator	Description	Syntax
!	NOT operator	`! expr1`
&&	AND operator	`expr1 && expr2`
||	OR operator	`expr1 || expr2`

Table 116. Relational Operators

You use relational operators on counters or status flags.

Operator	Description	Syntax
>	Greater than	`<identifier> > <numerical_value>`
>=	Greater than or Equal to	`<identifier> >= <numerical_value>`
==	Equals	`<identifier> == <numerical_value>`
!=	Does not equal	`<identifier> != <numerical_value>`
<=	Less than or equal to	`<identifier> <= <numerical_value>`
<	Less than	`<identifier> < <numerical_value>`

Notes to table:
1. <*identifier*> indicates a counter or status flag.
2. <*numerical_value*> indicates an integer.

Note:

- The <*boolean_expression*> in an `if` statement can contain a single event or multiple event conditions.
- When the boolean expression evaluates TRUE, the logic analyzer evaluates all the commands in the <*action_list*> concurrently.

14.4.5.3.3 <action_list>

List of actions that the Logic Analyzer performs within a state once a condition is satisfied.

- Each action must end with a semicolon (`;`).
- If you specify more than one action within an `if` or an `else if` clause, you must delimit the `action_list` with `begin` and `end` tokens.

Possible actions include:

Resource Manipulation Action

The resources the trigger flow description uses can be either counters or status flags.

Table 117. Resource Manipulation Actions

Action	Description	Syntax
`increment`	Increments a counter resource by 1	`increment <counter_identifier>;`
`decrement`	Decrements a counter resource by 1	`decrement <counter_identifier>;`
`reset`	Resets counter resource to initial value	`reset <counter_identifier>;`
`set`	Sets a status flag to 1	`set <register_flag_identifier>;`
`clear`	Sets a status flag to 0	`clear <register_flag_identifier>;`

Buffer Control Actions

Actions that control the acquisition buffer.

Table 118. Buffer Control Actions

Action	Description	Syntax
`trigger`	Stops the acquisition for the current buffer and ends analysis. This command is required in every flow definition.	`trigger <post-fill_count>;`
`segment_trigger`	Available only in segmented acquisition mode. Ends acquisition of the current segment. After evaluating this command, the Signal Tap Logic Analyzer starts acquiring from the next segment. If all segments are written, the Logic Analyzer	`segment_trigger <post-fill_count>;`

continued...

Action	Description	Syntax
	overwrites the oldest segment with the latest sample. When a trigger action is evaluated the acquisition stops.	
`start_store`	Active only in state-based storage qualifier mode. Asserts the `write_enable` to the Signal Tap acquisition buffer.	`start_store`
`stop_store`	Active only in state-based storage qualifier mode. De-asserts the `write_enable` signal to the Signal Tap acquisition buffer.	`stop_store`

Both `trigger` and `segment_trigger` actions accept an optional `post-fill_count` argument.

Related Links

Post-fill Count on page 373

State Transition Action

Specifies the next state in the custom state control flow. The syntax is:
`goto` *<state_label>*;

14.4.5.4 Using the State-Based Storage Qualifier Feature

Selecting a state-based storage qualifier type enables the `start_store` and `stop_store` actions. When you use these actions in conjunction with the expressions of the State-based trigger flow, you get maximum flexibility to control data written into the acquisition buffer.

Note: You can only apply the `start_store` and `stop_store` commands to a non-segmented buffer.

The `start_store` and `stop_store` commands are similar to the start and stop conditions of the **start/stop** storage qualifier mode. If you enable storage qualification, Signal Tap Logic Analyzer doesn't write data into the acquisition buffer until the `start_store` command occurs. However, in the state-based storage qualifier type you must include a `trigger` command as part of the trigger flow description. This `trigger` command is necessary to complete the acquisition and display the results on the waveform display.

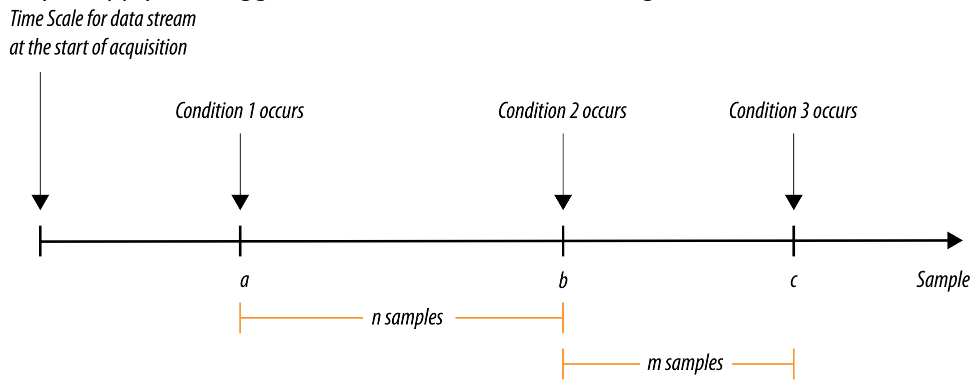
14.4.5.4.1 Storage Qualification Feature for the State-Based Trigger Flow.

This trigger flow description contains three trigger conditions that happen at different times after you click **Start Analysis**:

```
State 1: ST1:
    if ( condition1 )
       start_store;
    else if ( condition2 )
       trigger value;
    else if ( condition3 )
       stop_store;
```

Figure 196. Capture Scenario for Storage Qualification with the State-Based Trigger Flow

When you apply the trigger flow to the scenario in the figure:



1. The Signal Tap Logic Analyzer does not write into the acquisition buffer until **Condition 1** occurs (sample **a**).
2. When **Condition 2** occurs (sample **b**), the logic analyzer evaluates the `trigger value` command, and continues to write into the buffer to finish the acquisition.
3. The trigger flow specifies a `stop_store` command at sample **c**, which occurs `m` samples after the trigger point.
4. If the data acquisition finishes the post-fill acquisition samples before **Condition 3** occurs, the logic analyzer finishes the acquisition and displays the contents of the waveform. In this case, the capture ends if the post-fill count value is < `m`.
5. If the post-fill count value in the Trigger Flow description 1 is > `m` samples, the buffer pauses acquisition indefinitely, provided there is no recurrence of Condition 1 to trigger the logic analyzer to start capturing data again.

The Signal Tap Logic Analyzer continues to evaluate the `stop_store` and `start_store` commands even after evaluating the trigger. If the acquisition paused, click **Stop Analysis** to manually stop and force the acquisition to trigger. You can use counter values, flags, and the State diagram to help you perform the trigger flow. The counter values, flags, and the current state update in real-time during a data acquisition.

Example 36. Real data acquisition of the previous scenario

Figure 197. Storage Qualification with Post-Fill Count Value Less than m (Acquisition Successfully Completes)

The data capture finishes successfully. It uses a buffer with a sample depth of 64, *m = n = 10* , and `post-fill count = 5`.

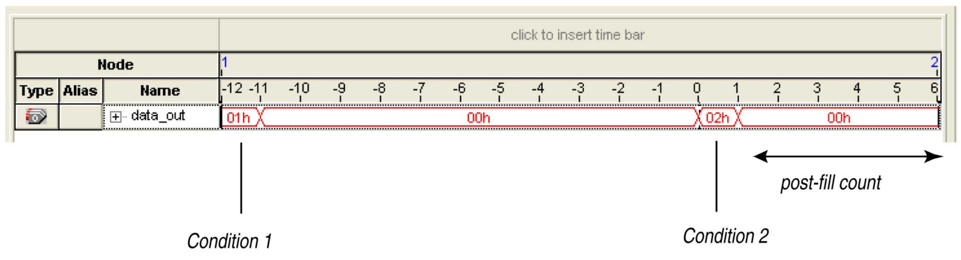
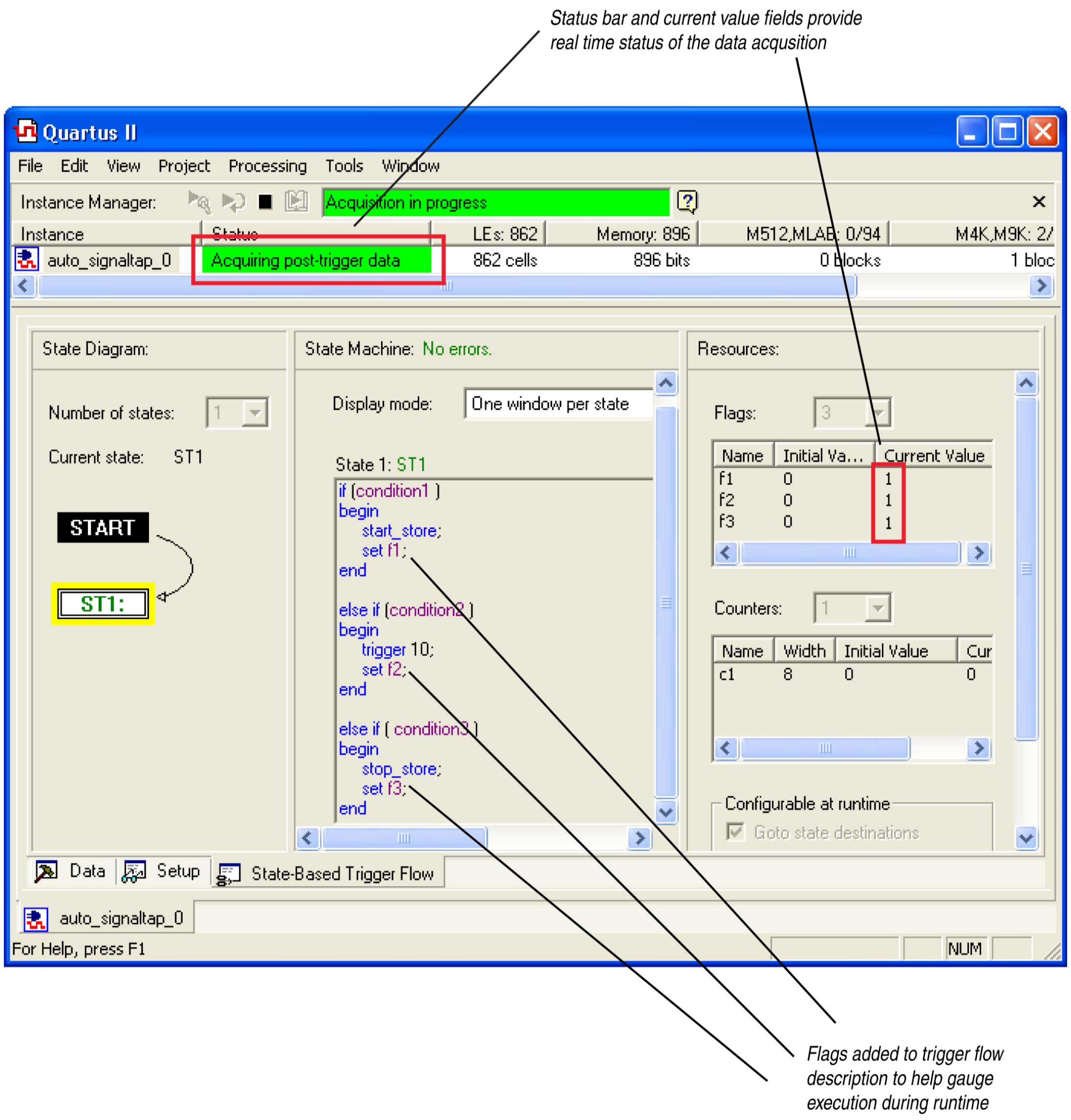


Figure 198. Storage Qualification with Post-Fill Count Value Greater than m (Acquisition Indefinitely Paused)

The logic analyzer pauses indefinitely, even after a trigger condition occurs due to a `stop_store` condition. This scenario uses a sample depth of 64, with `m = n = 10` and `post-fill count = 15`.


Status bar and current value fields provide real time status of the data acqusition
Flags added to trigger flow description to help gauge execution during runtime
Quartus II
File Edit View Project Processing Tools Window
Instance Manager:
Acquisition in progress
Data
Setup
State-Based Trigger Flow
For Help, press F1
NUM

Instance	Status	LEs: 862	Memory: 896	M512,MLAB: 0/94	M4K,M9K: 2/
auto_signaltap_0	Acquiring post-trigger data	862 cells	896 bits	0 blocks	1 bloc

State Diagram:

Number of states: 1

Current state: ST1

START → ST1:

State Machine: No errors.

Display mode: One window per state

State 1: ST1

```
if (condition1 )
begin
    start_store;
    set f1;
end

else if (condition2 )
begin
    trigger 10;
    set f2;
end

else if ( condition3 )
begin
    stop_store;
    set f3;
end
```

Resources:

Flags: 3

Name	Initial Va...	Current Value
f1	0	1
f2	0	1
f3	0	1

Counters: 1

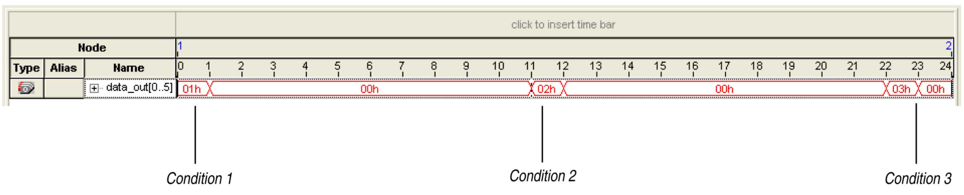
Name	Width	Initial Value	Cur
c1	8	0	0

Configurable at runtime

☑ Goto state destinations

auto_signaltap_0

Figure 199. Waveform After Forcing the Analysis to Stop



The combination of using counters, Boolean and relational operators in conjunction with the `start_store` and `stop_store` commands can give a clock-cycle level of resolution to controlling the samples that are written into the acquisition buffer.

Example 37. Trigger flow description that skips three clock cycles of samples after hitting condition 1

Code:

```
State 1: ST1
    start_store
    if ( condition1 )
    begin
       stop_store;
       goto ST2;
    end
State 2: ST2
    if (c1 < 3)
       increment c1; //skip three clock cycles; c1 initialized to 0
    else if (c1 == 3)
    begin
       start_store;//start_store necessary to enable writing to finish
                     //acquisition
       trigger;
    end
```

The figures show the data transaction on a continuous capture and the data capture when you apply the Trigger flow description.

Figure 200. Continuous Capture of Data Transaction

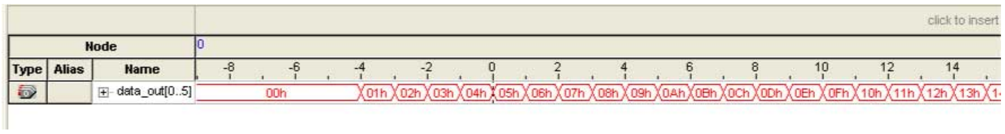
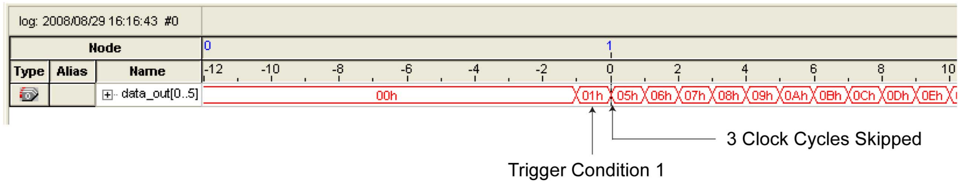


Figure 201. Capture of Data Transaction with Trigger Flow Description Applied



14.4.6 Specify Trigger Position

You can specify the amount of data the Logic Analyzer acquires before and after a trigger event. Positions for Runtime and Power-Up triggers are separate.

Signal Tap Logic Analyzer offers three pre-defined ratios of pre-trigger data to post-trigger data:

- **Pre**—Saves signal activity that occurred after the trigger (12% pre-trigger, 88% post-trigger).
- **Center**—Saves 50% pre-trigger and 50% post-trigger data.
- **Post**—Saves signal activity that occurred before the trigger (88% pre-trigger, 12% post-trigger).

These pre-defined ratios apply to both non-segmented buffers and segmented buffers.

Related Links

State-Based Triggering on page 363

14.4.6.1 Post-fill Count

In a custom state-based triggering flow with the `segment_trigger` and `trigger` buffer control actions, you can use the `post-fill_count` argument to specify a custom trigger position.

- If you do not use the `post-fill_count` argument, the trigger position for the affected buffer defaults to the trigger position you specified in the **Setup** tab.
- In the `trigger` buffer control action (for non-segmented buffers), `post-fill_count` specifies the number of samples to capture before stopping data acquisition.
- In the `segment_trigger` buffer control action (for segmented buffer), `post-fill_count` specifies a data segment.

  Note: In the case of `segment_trigger`, acquisition of the current buffer stops immediately if a subsequent triggering action is issued in the next state, regardless of the current buffer's post-fill count. The Logic Analyzer discards the remaining unfilled post-count acquisitions in the current buffer, and displays them as grayed-out samples in the data window.

When the Signal Tap data window displays the captured data, the trigger position appears as the number of post-count samples from the end of the acquisition segment or buffer.

Sample Number of Trigger Position = (*N* – *Post-Fill Count*)

In this case, *N* is the sample depth of either the acquisition segment or non-segmented buffer.

Related Links

Buffer Control Actions on page 368

14.4.7 Create a Power-Up Trigger

Power-up triggers capture events that occur during device initialization, immediately after you power or reset the FPGA.

The typical use of Signal Tap Logic Analyzer is triggering events that occur during normal device operation. You start an analysis manually once the target device is fully powered on and the JTAG connection for the device is available. With Signal Tap Power-Up Trigger feature, the Signal Tap Logic Analyzer captures data immediately after device initialization.

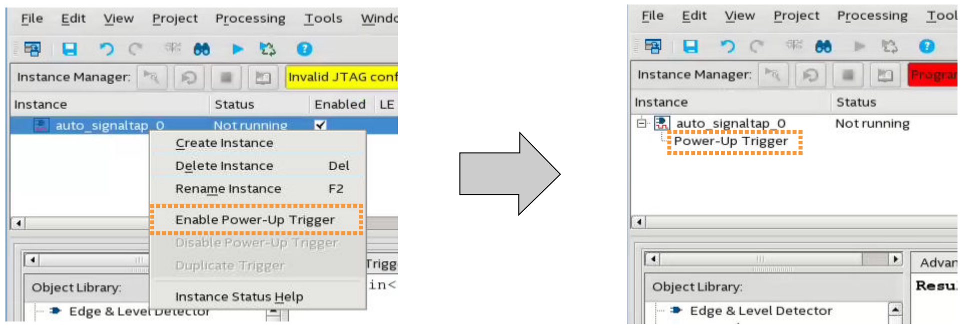
You can add a different Power-Up Trigger to each logic analyzer instance in the **Signal Tap Instance Manager** pane.

14.4.7.1 Enabling a Power-Up Trigger

To enable the Power-Up Trigger for a logic analyzer instance, right-click the instance and click **Enable Power-Up Trigger**. Alternatively, click **Edit ➤ Enable Power-Up Trigger**.

Power-Up Trigger appears as a child instance below the name of the selected instance. The node list displays the default trigger conditions.

Figure 202. Enabling Power-Up Trigger in Signal Tap Logic Analyzer Editor



To disable a Power-Up Trigger, right-click the instance and click **Disable Power-Up Trigger**.

14.4.7.2 Manage and Configure Power-Up and Runtime Trigger Conditions

You can create basic, comparison, and advanced trigger conditions for your enabled Power-Up Trigger as you do with a Run-Time Trigger.

Since each instance now has two sets of trigger conditions—the Power-Up Trigger and the Run-Time Trigger—you can differentiate between the two with color coding. Power-Up Trigger conditions that you can adjust are color coded light blue, while Run-Time Trigger conditions you cannot adjust remain white.

To switch between the trigger conditions of the Power-Up Trigger and the Run-Time Trigger, double-click the instance name or the Power-Up Trigger name in the **Instance Manager**.

You cannot make changes to Power-Up Trigger conditions that would normally require a full recompile with Runtime Trigger conditions, such as adding signals, deleting signals, or changing between basic, comparison, and advanced triggers. To apply these changes to the Power-Up Trigger conditions, first make the changes using the Runtime Trigger conditions.

Note: Any change made to the Power-Up Trigger conditions requires that you recompile the Signal Tap Logic Analyzer instance, even if a similar change to the Runtime Trigger conditions does not require a recompilation.

To copy trigger conditions from a Run-Time Trigger to a Power-Up Trigger or vice versa, right-click the trigger name in the **Instance Manager** and click **Duplicate Trigger**. Alternatively, select the trigger name and click **Edit ➤ Duplicate Trigger**.

You can also use In-System Sources and Probes in conjunction with the Signal Tap Logic Analyzer to force trigger conditions. The In-System Sources and Probes feature allows you to drive and sample values on to selected nets over the JTAG chain.

Related Links

Design Debugging Using In-System Sources and Probes on page 49

14.4.8 External Triggers

To trigger Signal Tap Logic Analyzer from an external source, you can create an external trigger input.

The external trigger input behaves like trigger condition 1, in that it must evaluate to TRUE before the logic analyzer evaluates any other configured trigger conditions.

Signal Tap Logic Analyzer supplies a signal to trigger external devices or other logic analyzer instances. These features allow you to synchronize external logic analysis equipment with the internal logic analyzer. Power-Up Triggers can use the external triggers feature, but they must use the same source or target signal as their associated Run-Time Trigger.

You can use external triggers to perform cross-triggering on a hard processor system (HPS):

- Use your processor debugger to configure the HPS to obey or disregard cross-trigger request from the FPGA, and to issue or not issue cross-trigger requests to the FPGA.
- Use your processor debugger in combination with the Signal Tap external trigger feature to develop a dynamic combination of cross-trigger behaviors.
- You can use the cross-triggering feature with the ARM Development Studio 5 (DS-5) software to implement a system-level debugging solution for your Intel FPGA SoC.

Related Links

- FPGA-Adaptive Software Debug and Performance Analysis white paper
- Signal Configuration Pane
  In *Intel Quartus Prime Help*

14.4.8.1 Using the Trigger Out of One Analyzer as the Trigger In of Another Analyzer

An advanced feature of the Signal Tap Logic Analyzer is the ability to use the **Trigger out** of one analyzer as the **Trigger in** to another analyzer. This feature allows you to synchronize and debug events that occur across multiple clock domains.

To perform this operation, first turn on **Trigger out** for the source logic analyzer instance. On the **Instance** list of the **Trigger out** trigger, select the targeted logic analyzer instance. For example, if the instance named `auto_signaltap_0` should trigger `auto_signaltap_1`, select `auto_signaltap_1|trigger_in` .

Turning on **Trigger out** automatically enables the **Trigger in** of the targeted logic analyzer instance and fills in the **Instance** field of the **Trigger in** trigger with the **Trigger out** signal from the source logic analyzer instance. In this example, `auto_signaltap_0` is targeting `auto_signaltap_1`. The Trigger In **Instance** field of `auto_signaltap_1` is automatically filled in with `auto_signaltap_0|trigger_out`.

14.5 Compiling the Design

To incorporate the Signal Tap logic in your design and enable the JTAG connection, you must compile your project. When you add a `.stp` file to your project, the Signal Tap Logic Analyzer becomes part of your design. When you debug your design with a traditional external logic analyzer, you must often make changes to the signals you want to monitor as well as the trigger conditions.

Note: Because these adjustments require that you recompile your design when using the Signal Tap Logic Analyzer, use the Signal Tap Logic Analyzer feature along with incremental compilation in the Intel Quartus Prime software to reduce recompilation time.

14.5.1 Faster Compilations with Intel Quartus Prime Incremental Compilation

You can add a Signal Tap Logic Analyzer instance to your design without recompiling your original source code. Incremental compilation enables you to preserve the synthesis and fitting results of your original design.

When you compile your design including a `.stp` file, Intel Quartus Prime software automatically adds the `sld_signaltap` and `sld_hub` entities to the compilation hierarchy. These two entities are the main components of the Signal Tap Logic Analyzer, providing the trigger logic and JTAG interface required for operation.

Incremental compilation is also useful when you want to modify the configuration of the `.stp` file. For example, you can change the buffer sample depth or memory type without performing a full compilation. Instead, you only recompile the Signal Tap Logic Analyzer, configured as its own design partition.

14.5.1.1 Enabling Incremental Compilation for Your Design

When enabled for your design, the Signal Tap Logic Analyzer is always a separate partition. After the first compilation, you can use the Signal Tap Logic Analyzer to analyze signals from the post-fit netlist. If your partitions are designed correctly, subsequent compilations due to Signal Tap Logic Analyzer settings take less time.

The netlist type for the top-level partition defaults to **source**. To take advantage of incremental compilation, specify the Netlist types for the partitions you want to tap as **Post-fit**.

Related Links

Intel Quartus Prime Incremental Compilation for Hierarchical and Team-Based Design documentation

14.5.1.2 Using Incremental Compilation with the Signal Tap Logic Analyzer

The Signal Tap Logic Analyzer uses the incremental compilation flow by default. For all signals that you want to connect to the Signal Tap Logic Analyzer from the post-fit netlist:

1. In the Design Partitions window, set the netlist type of the partition that contains the signals to **Post-Fit**, with a Fitter Preservation Level of **Placement and Routing**.
2. In the **Node Finder**, use the **Signal Tap: post-fitting filter** to add the signals of interest to your Signal Tap configuration file.
3. If you want to add signals from the pre-synthesis netlist, set the netlist type to **Source File** and use the **Signal Tap: pre-synthesis filter** in the **Node Finder**. Do not use the netlist type **Post-Synthesis** with the Signal Tap Logic Analyzer.

Caution: When using post-fit and pre-synthesis nodes:

- Read all incremental compilation guidelines to ensure the proper partitioning of a project.
- To speed up compile time, use only post-fit nodes for partitions specified as preservation-level post-fit.
- Do not mix pre-synthesis and post-fit nodes in any partition. If you must tap pre-synthesis nodes for a particular partition, make all tapped nodes in that partition pre-synthesis nodes and change the netlist type to **source** in the design partitions window.

Node names can differ between a pre-synthesis netlist and a post-fit netlist. In general, registers and user input signals share common names between the two netlists. During compilation, certain optimizations change the names of combinational signals in your RTL. If the type of node name chosen does not match the netlist type, the compiler may not be able to find the signal to connect to your Signal Tap Logic Analyzer instance for analysis. The compiler issues a critical warning to alert you of this scenario. The signal that is not connected is tied to ground in the **Signal Tap data** tab.

If you do use incremental compilation flow with the Signal Tap Logic Analyzer and source file changes are necessary, be aware that you may have to remove compiler-generated post-fit net names. Source code changes force the affected partition to go through resynthesis. During synthesis, the compiler cannot find compiler-generated net names from a previous compilation.

Note: Intel FPGA recommends using only registered and user-input signals as debugging taps in your `.stp` whenever possible.

Both registered and user-supplied input signals share common node names in the pre-synthesis and post-fit netlist. As a result, using only registered and user-supplied input signals in your `.stp` limits the changes you need to make to your Signal Tap Logic Analyzer configuration.

You can check the nodes that are connected to each Signal Tap instance using the In-System Debugging compilation reports. These reports list each node name you selected to connect to a Signal Tap instance, the netlist type used for the particular connection, and the actual node name used after compilation. If the incremental compilation flow is not used, the In-System Debugging reports are located in the Analysis & Synthesis folder. If the incremental compilation flow is used, this report is located in the Partition Merge folder.

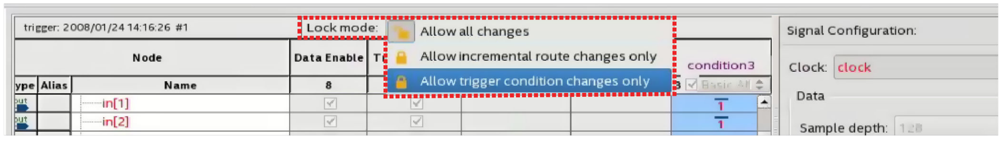
To verify that your original design was not modified, examine the messages in the **Partition Merge** section of the Compilation Report.

Unless you make changes to your design partitions that require recompilation, only the Signal Tap design partition is recompiled. If you make subsequent changes to only the `.stp`, only the Signal Tap design partition must be recompiled, reducing your recompilation time.

14.5.2 Prevent Changes Requiring Recompilation

Configure the `.stp` to prevent changes that normally require recompilation. To do this, select a **Lock mode** from above the node list in the **Setup** tab. To lock your configuration, choose **Allow trigger condition changes only**.

Figure 203. Allow Trigger Conditions Change Only



Related Links

Verify Whether You Need to Recompile Your Project on page 382

14.5.3 Incremental Route with Rapid Recompile

You can use Incremental Route with Rapid Recompile to decrease compilation times. After performing a full compilation on your design, you can use the Incremental Route flow to achieve a 2-4x speedup over a flat compile. The Incremental Route flow is not compatible with Partial Reconfiguration.

Intel Quartus Prime Standard Edition software supports Incremental Route with Rapid Recompile for Arria V, Cyclone V, and Stratix V devices.

Related Links

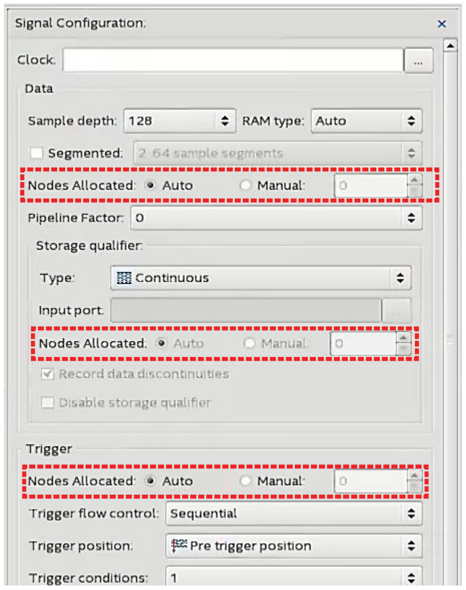
Running Rapid Recompile
In *Intel Quartus Prime Pro Edition Handbook Volume 1*

14.5.3.1 Using the Incremental Route Flow

To use the Incremental Route flow:

1. Open your design and run **Analysis & Elaboration** (or a full compilation) to give node visibility in Signal Tap.
2. Add Signal Tap to your design.
3. In the Signal Tap **Signal Configuration** pane, specify **Manual** in the **Nodes Allocated** field for Trigger and Data nodes (and Storage Qualifier, if used).

Figure 204. Manually Allocate Nodes



Signal Configuration:

Clock:

Data

Sample depth: 128 RAM type: Auto

Segmented: 2 64 sample segments

Nodes Allocated: ● Auto ○ Manual: 0

Pipeline Factor: 0

Storage qualifier:

Type: Continuous

Input port:

Nodes Allocated: ● Auto ○ Manual: 0

✓ Record data discontinuities

Disable storage qualifier

Trigger

Nodes Allocated: ● Auto ○ Manual: 0

Trigger flow control: Sequential

Trigger position: Pre trigger position

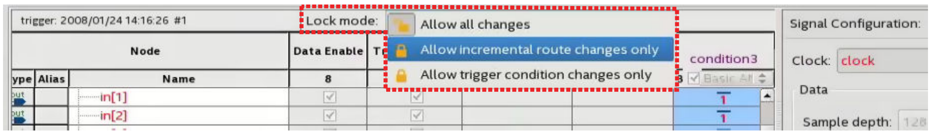
Trigger conditions: 1

Manual node allocation allows you to control the number of nodes compiled into the design, which is critical for the Incremental Route flow.

When you select **Auto** allocation, the number of nodes compiled into the design matches the number of nodes in the **Setup** tab. If you add a node later, you create a mismatch between the amount of nodes the device requires and the amount of compiled nodes, and you must perform a full compilation.

4. Specify the number of nodes that you estimate necessary for the debugging process. You can increase the number of nodes later, but this requires more compilation time.
5. Add the nodes that you want to tap.
6. If you have not fully compiled your project, run a full compilation. Otherwise, start incremental compile using Rapid Recompile.
7. Debug and determine additional signals of interest.
8. (Optional) Select **Allow incremental route changes only** lock-mode.

Figure 205. Incremental Route Lock-Mode



trigger: 2008/01/24 14:16:26 #1

Lock mode:
- Allow all changes
- Allow incremental route changes only
- Allow trigger condition changes only

Node			Data Enable	condition3
Type	Alias	Name	8	
		in[1]	✓	
		in[2]	✓	

Signal Configuration:

Clock: clock

Data

Sample depth: 128

9. Add additional nodes in the Signal Tap **Setup** tab.
   — Do not exceed the number of manually allocated nodes you specified.
   — Avoid making changes to non-runtime configurable settings.
10. Click the Rapid Recompile icon from the toolbar. Alternatively, click **Processing ➤ Start Rapid Recompile**.

Note: The previous steps set up your design for Incremental Route, but the actual Incremental Route process begins when you perform a Rapid Recompile.

14.5.3.2 Tips to Achieve Maximum Speedup

- Basic AND (which applies to Storage Qualifier as well as trigger input) is the fastest for the Incremental Route flow.
- Basic OR is slower for the Incremental Route flow, but if you avoid changing the parent-child relationship of nodes within groups, you can minimize the impact on compile time. You can change the sibling relationships of nodes.
  — Basic OR and advanced triggers require re-synthesis when you change the number/names of tapped nodes.
- Use the Incremental Route lock-mode to avoid inadvertent changes requiring a full compilation.

14.5.4 Timing Preservation with the Signal Tap Logic Analyzer

In addition to verifying functionality, timing closure is one of the most crucial processes in successful operation of your design.

Note: When you compile a project with a Signal Tap Logic Analyzer without the use of incremental compilation, you must add IP to your existing design. This addition often impacts the existing placement, routing, and timing of your design. To minimize the effect that the Signal Tap Logic Analyzer has on your design, use incremental compilation for your project. Incremental compilation is the default setting in new designs. You can easily enable incremental compilation in existing designs. When the Signal Tap Logic Analyzer is in a design partition, it has little to no affect on your design.

For Intel Arria 10 devices, the Intel Quartus Prime Standard Edition software does not support timing preservation for post-fit taps with Rapid Recompile.

Use the following techniques to help maintain timing:

- Avoid adding critical path signals to your `.stp`.
- Minimize the number of combinational signals you add to your `.stp`, and add registers whenever possible.
- Specify an f_{MAX} constraint for each clock in your design.

Related Links

Timing Closure and Optimization
In *Intel Quartus Prime Standard Edition Handbook Volume 2*

14.5.5 Performance and Resource Considerations

When you perform logic analysis of your design, you can see the necessary trade-off between runtime flexibility, timing performance, and resource usage.

The Signal Tap Logic Analyzer allows you to select runtime configurable parameters to balance the need for runtime flexibility, speed, and area.

The default values of the runtime configurable parameters provide maximum flexibility, so you can complete debugging as quickly as possible; however, you can adjust these settings to determine whether there is a more appropriate configuration for your design. Because performance results are design-dependent, try these options in different combinations until you achieve the desired balance between functionality, performance, and utilization.

14.5.5.1 Signal Tap Logic in Critical Path

If Signal Tap logic is part of your critical path, follow these tips to speed up the performance of the Signal Tap Logic Analyzer:

- **Disable runtime configurable options**—Certain resources are allocated to accommodate for runtime flexibility. If you use either advanced triggers or State-based triggering flow, disable runtime configurable parameters for a boost in f_{MAX} of the Signal Tap logic.
  - — If you are using State-based triggering flow, try disabling the **Goto state destination** option and performing a recompilation before disabling the other runtime configurable options. The **Goto state destination** option has the greatest impact on f_{MAX}, as compared to the other runtime configurable options.
- **Minimize the number of signals that have Trigger Enable selected**—By default, Signal Tap Logic Analyzer enable the **Trigger Enable** option for all signals that you add to the `.stp` file. For signals that you do not plan to use as triggers, turn this option off.
- **Turn on Physical Synthesis for register retiming**—If many (more than the number of inputs that fit in a LAB) enabled triggering signals fan-in logic to a gate-based triggering condition (basic trigger condition or a logical reduction operator in the advanced trigger tab), turn on **Perform register retiming**. This can help balance combinational logic across LABs.

14.5.5.2 Signal Tap Logic Using Critical Resources

If your design is resource constrained, follow these tips to reduce the logic or memory the Signal Tap Logic Analyzer uses:

- **Disable runtime configurable options**—Disabling runtime configurability for advanced trigger conditions or runtime configurable options in the State-based triggering flow results in fewer LEs.
- **Minimize the number of segments in the acquisition buffer**—You can reduce the logic resources that the Signal Tap Logic Analyzer uses if you limit the segments in your sampling buffer
- **Disable the Data Enable for signals that you use only for triggering**—By default, Signal Tap Logic Analyzer enables **data enable** options for all signals. Turning off the **data enable** option for signals you use only as trigger inputs saves on memory resources.

14.6 Program the Target Device or Devices

After you add the Signal Tap Logic Analyzer to your project and re-compile, you can configure the FPGA target device.

If you want to debug multiple designs simultaneously, configure the device from the `.stp` instead of the Intel Quartus Prime Programmer. This allows you to open more than one `.stp` file and program multiple devices.

14.6.1 Ensure Setting Compatibility Between .stp and .sof Files

A `.stp` file is compatible with a `.sof` file when the settings for the logic analyzer, such as the size of the capture buffer and the signals you use for monitoring or triggering, match the programming settings of the target device. If the files are not compatible, you can still program the device, but you cannot run or control the logic analyzer from the Signal Tap Logic Analyzer Editor.

- To ensure programming compatibility, program your device with the `.sof` file generated in the most recent compilation.
- To check whether a particular `.sof` is compatible with the current Signal Tap configuration, attach the `.sof` to the SOF manager.

Note: When the Signal Tap Logic Analyzer detects incompatibility after the analysis starts, the Intel Quartus Prime software generates a system error message containing two CRC values: the expected value and the value retrieved from the `.stp` instance on the device. The CRC value comes from all Signal Tap settings that affect the compilation.

Although having a Intel Quartus Prime project is not required when using an `.stp`, it is recommended. The project database contains information about the integrity of the current Signal Tap Logic Analyzer session. Without the project database, there is no way to verify that the current `.stp` file matches the `.sof` file in the device. If you have an `.stp` file that does not match the `.sof` file, the Signal Tap Logic Analyzer can capture incorrect data.

Related Links

Manage Multiple Signal Tap Files and Configurations on page 350

14.6.2 Verify Whether You Need to Recompile Your Project

Before starting a debugging session, do not make any changes to the `.stp` settings that require recompiling the project.

To verify whether a change you made requires recompiling the project, check the Signal Tap status display at the top of the **Instance Manager** pane. This feature allows you to undo the change, so that you do not need to recompile your project.

Related Links

Prevent Changes Requiring Recompilation on page 378

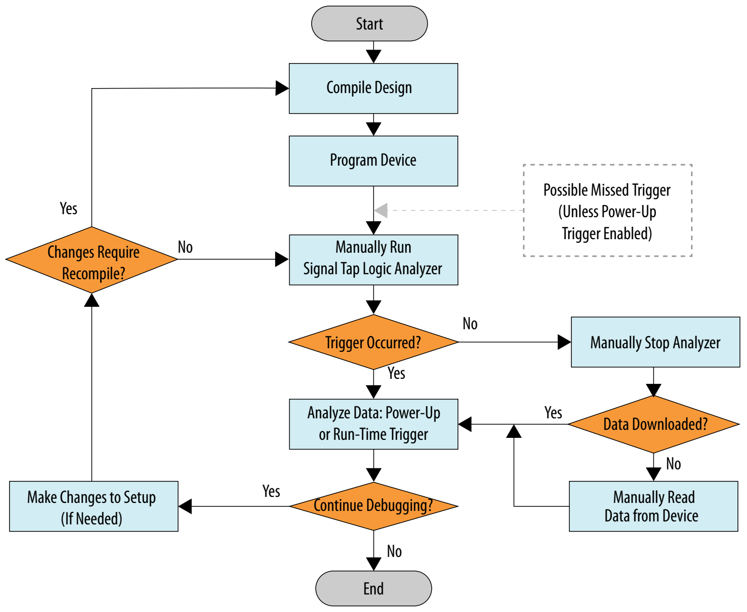
14.7 Running the Signal Tap Logic Analyzer

Debugging Signal Tap Logic Analyzer is similar using an external logic analyzer. You initialize the logic analyzer by starting an analysis. When your trigger event occurs, the logic analyzer stores the captured data in the device's memory buffer, and then transfers this data to the `.stp` file with the JTAG connection.

You can also perform the equivalent of a force trigger instruction that lets you view the captured data currently in the buffer without a trigger event occurring.

The flowchart shows how you operate the Signal Tap Logic Analyzer. indicates where Power-Up and Runtime Trigger events occur and when captured data from these events is available for analysis.

Figure 206. Power-Up and Runtime Trigger Events Flowchart


Start
Compile Design
Program Device
Possible Missed Trigger (Unless Power-Up Trigger Enabled)
Yes
Changes Require Recompile?
No
Manually Run Signal Tap Logic Analyzer
Trigger Occurred?
No
Manually Stop Analyzer
Yes
Analyze Data: Power-Up or Run-Time Trigger
Yes
Data Downloaded?
No
Manually Read Data from Device
Make Changes to Setup (If Needed)
Yes
Continue Debugging?
No
End

You can also use In-System Sources and Probes in conjunction with the Signal Tap Logic Analyzer to force trigger conditions. The In-System Sources and Probes feature allows you to drive and sample values on to selected signals over the JTAG chain.

Related Links

Design Debugging Using In-System Sources and Probes on page 49

14.7.1 Runtime Reconfigurable Options

When you use Runtime Trigger mode, you can change certain settings in the `.stp` without recompiling your design.

Table 119. Runtime Reconfigurable Features

Runtime Reconfigurable Setting	Description
Basic Trigger Conditions and Basic Storage Qualifier Conditions	You can change without recompiling all signals that have the Trigger condition turned on to any basic trigger condition value
Comparison Trigger Conditions and Comparison Storage Qualifier Conditions	All the comparison operands, the comparison numeric values, and the interval bound values are runtime-configurable. You can also switch from Comparison to Basic OR trigger at runtime without recompiling.
Advanced Trigger Conditions and Advanced Storage Qualifier Conditions	Many operators include runtime configurable settings. For example, all comparison operators are runtime-configurable. Configurable settings appear with a white background in the block representation. This runtime reconfigurable option is turned on in the **Object Properties** dialog box.
Switching between a storage-qualified and a continuous acquisition	Within any storage-qualified mode, you can switch to continuous capture mode without recompiling the design. To enable this feature, turn on **disable storage qualifier**.
State-based trigger flow parameters	Refer to *Runtime Reconfigurable Settings, State-Based Triggering Flow*

Runtime Reconfigurable options can save time during the debugging cycle by allowing you to cover a wider possible scenario of events without the need to recompile the design. You may experience a slight impact to the performance and logic utilization. You can turn off runtime re-configurability for advanced trigger conditions and the state-based trigger flow parameters, boosting performance and decreasing area utilization.

To configure the `.stp` file to prevent changes that normally require recompilation in the **Setup** tab, select **Allow Trigger Condition changes only** above the node list.

In Incremental Route lock mode, **Allow incremental route changes only**, limits to changes that only require an Incremental Route compilation, and not a full compile.

This example illustrates a potential use case for Runtime Reconfigurable features, by providing a storage qualified enabled State-based trigger flow description, and showing how to modify the size of a capture window at runtime without a recompile. This example gives you equivalent functionality to a segmented buffer with a single trigger condition where the segment sizes are runtime reconfigurable.

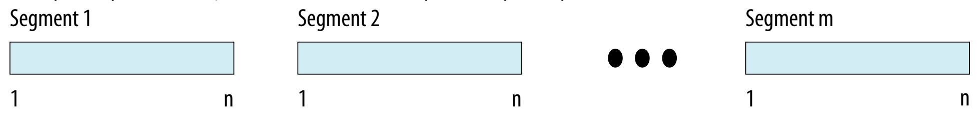
```
state ST1:
if ( condition1 && (c1 <= m) )// each "segment"  triggers on condition
                              // 1
begin                           // m  = number of total "segments"
   start_store;
   increment c1;
   goto ST2:
End

else (c1 > m )                // This else condition handles the last
                              // segment.
begin
    start_store
    Trigger (n-1)
end

state ST2:
if ( c2 >= n)                //n = number of samples to capture in each
                             //segment.
begin
    reset c2;
    stop_store;
    goto ST1;
end

else (c2 < n)
begin
    increment c2;
    goto ST2;
end
```

Note: *m* x *n* must equal the sample depth to efficiently use the space in the sample buffer.

The next figure shows the segmented buffer that the trigger flow example describes.

Figure 207. Segmented Buffer Created with Storage Qualifier and State-Based Trigger

Total sample depth is fixed, where *m* x *n* must equal sample depth.



During runtime, you can modify the values m and n. Changing the m and n values in the trigger flow description adjust the segment boundaries without recompiling.

You can add states into the trigger flow description and selectively mask out specific states and enable other ones at runtime with status flags.

This example is like the previous example with an additional state inserted. You use this extra state to specify a different trigger condition that does not use the storage qualifier feature. You insert status flags into the conditional statements to control the execution of the trigger flow.

```
state ST1 :
    if (condition2  && f1)               // additional state added for a non-
segmented
                                        // acquisition set f1 to enable state
        begin
             start_store;
             trigger
        end
    else if (! f1)
        goto ST2;
state ST2:
    if ( (condition1 && (c1 <= m)  && f2) // f2 status flag used to mask
state. Set f2
                                          // to enable
        begin
           start_store;
           increment c1;
           goto ST3:
        end
    else (c1 > m )
           start_store
   Trigger (n-1)
    end
state ST3:
    if ( c2 >= n)
        begin
            reset c2;
            stop_store;
            goto ST1;
        end
    else (c2 < n)
    begin
        increment c2;
        goto ST2;
    end
```

14.7.2 Signal Tap Status Messages

The table describes the text messages that might appear in the Signal Tap Status Indicator in the **Instance Manager** pane before, during, and after a data acquisition. Use these messages to monitor the state of the logic analyzer or what operation it is performing.

Table 120. Text Messages in the Signal Tap Status Indicator

Message	Message Description
`Not running`	The Signal Tap Logic Analyzer is not running. There is no connection to a device or the device is not configured.
`(Power-Up Trigger) Waiting for clock` (1)	The Signal Tap Logic Analyzer is performing a Runtime or Power-Up Trigger acquisition and is waiting for the clock signal to transition.

continued...

Message	Message Description
`Acquiring (Power-Up) pre-trigger data` (1)	The trigger condition has not been evaluated yet. A full buffer of data is collected if using the non-segmented buffer acquisition mode and storage qualifier type is continuous.
`Trigger In conditions met`	Trigger In condition has occurred. The Signal Tap Logic Analyzer is waiting for the condition of the first trigger condition to occur. This can appear if Trigger In is specified.
`Waiting for (Power-up) trigger` (1)	The Signal Tap Logic Analyzer is now waiting for the trigger event to occur.
`Trigger level` <x> met	The condition of trigger condition *x* has occurred. The Signal Tap Logic Analyzer is waiting for the condition specified in condition x + 1 to occur.
`Acquiring (power-up) post-trigger data` (1)	The entire trigger event has occurred. The Signal Tap Logic Analyzer is acquiring the post-trigger data. The amount of post-trigger data collected is you define between 12%, 50%, and 88% when the non-segmented buffer acquisition mode is selected.
`Offload acquired (Power-Up) data` (1)	Data is being transmitted to the Intel Quartus Prime software through the JTAG chain.
`Ready to acquire`	The Signal Tap Logic Analyzer is waiting for you to initialize the analyzer.

1. This message can appear for both Runtime and Power-Up Trigger events. When referring to a Power-Up Trigger, the text in parentheses is added.

Note: In segmented acquisition mode, pre-trigger and post-trigger do not apply.

14.8 View, Analyze, and Use Captured Data

Use the Signal Tap Logic Analyzer interface to examine the data you captured manually or using a trigger, and use your findings to debug your design.

When in the Data view, you can use the drag-to-zoom feature by left-clicking to isolate the data of interest.

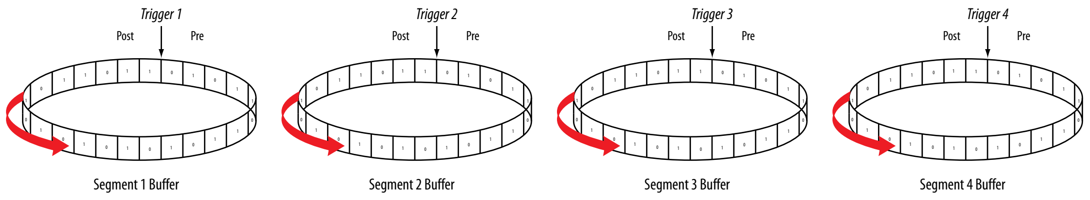
14.8.1 Capturing Data Using Segmented Buffers

Segmented Acquisition buffers allow you to perform multiple captures with a separate trigger condition for each acquisition segment. This feature allows you to capture a recurring event or sequence of events that span over a long period time efficiently.

Each acquisition segment acts as a non-segmented buffer, continuously capturing data when it is activated. When you run an analysis with the **segmented buffer** option enabled, the Signal Tap Logic Analyzer performs back-to-back data captures for each acquisition segment within your data buffer. You define the trigger flow, or the type and order in which the trigger conditions evaluate for each buffer, either in the Sequential trigger flow control or in the Custom State-based trigger flow control.

The following figure shows a segmented acquisition buffer with four segments represented as four separate non-segmented buffers.

Figure 208. Segmented Acquisition Buffer

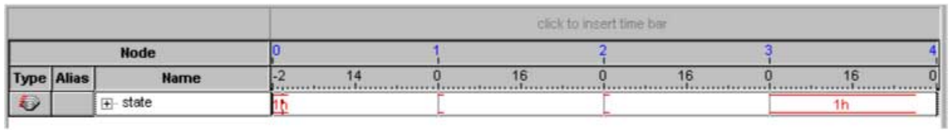


The Signal Tap Logic Analyzer finishes an acquisition with a segment, and advances to the next segment to start a new acquisition. Depending on when a trigger condition occurs, it may affect the way the data capture appears in the waveform viewer. The figure illustrates the data capture method. The Trigger markers—Trigger 1, Trigger 2, Trigger 3 and Trigger 4—refer to the evaluation of the `segment_trigger` and `trigger` commands in the Custom State-based trigger flow. If you use a sequential flow, the Trigger markers refer to trigger conditions specified within the **Setup** tab.

If the Segment 1 Buffer is the active segment and Trigger 1 occurs, the Signal Tap Logic Analyzer starts evaluating Trigger 2 immediately. Data Acquisition for Segment 2 buffer starts when either Segment Buffer 1 finishes its post-fill count, or when Trigger 2 evaluates as TRUE, whichever condition occurs first. Thus, trigger conditions associated with the next buffer in the data capture sequence can preempt the post-fill count of the current active buffer. This allows the Signal Tap Logic Analyzer to accurately capture all the trigger conditions that have occurred. Unused samples appear as a blank space in the waveform viewer.

The next figure shows an example of a capture using sequential flow control with the trigger condition for each segment specified as **Don’t Care**.

Figure 209. Segmented Capture with Preemption of Acquisition Segments



Each segment before the last captures only one sample, because the next trigger condition immediately preempts capture of the current buffer. The trigger position for all segments is specified as pre-trigger (10% of the data is before the trigger condition and 90% of the data is after the trigger position). Because the last segment starts immediately with the trigger condition, the segment contains only post-trigger data. The three empty samples in the last segment are left over from the pre-trigger samples that the Signal Tap Logic Analyzer allocated to the buffer.

For the sequential trigger flow, the **Trigger Position** option applies to every segment in the buffer. For maximum flexibility on defining the trigger position, use the custom state-based trigger flow. By adjusting the trigger position specific to your debugging requirements, you can help maximize the use of the allocated buffer space.

14.8.2 Differences in Pre-fill Write Behavior Between Different Acquisition Modes

The Signal Tap Logic Analyzer uses one of the following three modes when writing into the acquisition memory:

- **Non-segmented buffer**
- **Non-segmented buffer with a storage qualifier**
- **Segmented buffer**

There are subtle differences in the amount of data captured immediately after running the Signal Tap Logic Analyzer and before any trigger conditions occur. A non-segmented buffer, running in continuous mode, completely fills the buffer with sampled data before evaluating any trigger conditions. Thus, a non-segmented capture without any storage qualification enabled always shows a waveform with a full buffer's worth of data captured.

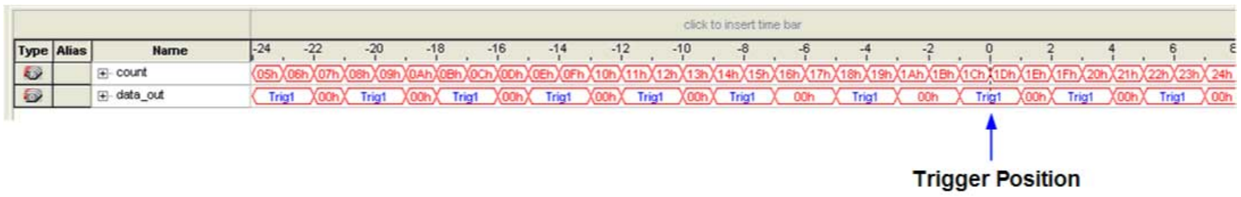
Filling the buffer provides you with as much data as possible within the capture window. The buffer gets pre-filled with data samples prior to evaluating the trigger condition. As such, Signal Tap requires that the buffer be filled at least once before any data can be retrieved through the JTAG connection and prevents the buffer from being dumped during the first acquisition prior to a trigger condition when you perform a **Stop Analysis**.

For segmented buffers and non-segmented buffers using any storage qualification mode, the Signal Tap Logic Analyzer immediately evaluates all trigger conditions while writing samples into the acquisition memory. This evaluation is especially important when using any storage qualification on the data set. The logic analyzer may miss a trigger condition if it waits to capture a full buffer's worth of data before evaluating any trigger conditions,

If the trigger event occurs on any data sample before the specified amount of pre-trigger data has occurred, then the Signal Tap Logic Analyzer triggers and begins filling memory with post-trigger data, regardless of the amount of pre-trigger data you specify. For example, if you set the trigger position to 50% and set the logic analyzer to trigger on a processor reset, start the logic analyzer, and then power on your target system, the logic analyzer triggers. However, the logic analyzer memory is filled only with post-trigger data, and not any pre-trigger data, because the trigger event, which has higher precedence than the capture of pre-trigger data, occurred before the pre-trigger condition was satisfied.

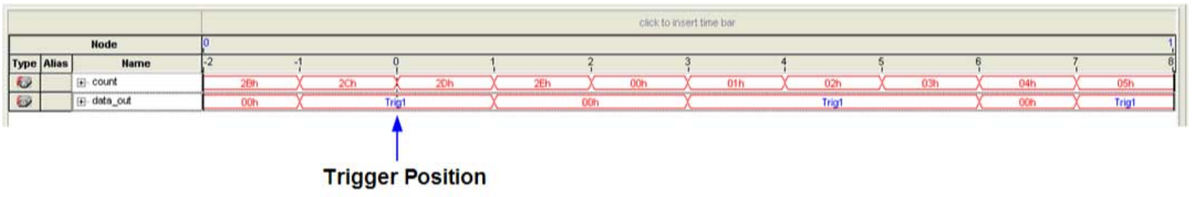
The figures for continuous data capture and conditional data capture show the difference between a non-segmented buffer in continuous mode and a non-segmented buffer using a storage qualifier. The configuration of the logic analyzer waveforms below is a base trigger condition, sample depth of 64 bits, and **Post trigger position**.

Figure 210. Signal Tap Logic Analyzer Continuous Data Capture


click to insert time bar
Type
Alias
Name
-24 -22 -20 -18 -16 -14 -12 -10 -8 -6 -4 -2 0 2 4 6 8
count
05h 06h 07h 08h 09h 0Ah 0Bh 0Ch 0Dh 0Eh 0Fh 10h 11h 12h 13h 14h 15h 16h 17h 18h 19h 1Ah 1Bh 1Ch 1Dh 1Eh 1Fh 20h 21h 22h 23h 24h
data_out
Trig1 00h Trig1 00h Trig1 00h Trig1 00h Trig1 00h Trig1 00h Trig1 00h Trig1 00h Trig1 00h Trig1 00h Trig1 00h
Trigger Position

In the continuous data capture, Trig1 occurs several times in the data buffer before the Signal Tap Logic Analyzer actually triggers. A full buffer's worth of data is captured before the logic analyzer evaluates any trigger conditions. After the trigger condition occurs, the logic analyzer continues acquisition until it captures eight additional samples (12% of the buffer, as defined by the "post-trigger" position).

Figure 211. Signal Tap Logic Analyzer Conditional Data Capture



Note to figure:

1. Conditional capture, storage always enabled, post-fill count.
2. Signal Tap Logic Analyzer capture of a recurring pattern using a non-segmented buffer in conditional mode. The configuration of the logic analyzer is a basic trigger condition "Trig1" and sample depth of 64 bits. The **Trigger in** condition is **Don't care**, which means that every sample is captured.

In conditional capture the logic analyzer triggers immediately. As in continuous capture, the logic analyzer completes the acquisition with eight samples, or 12% of 64, the sample capacity of the acquisition buffer.

14.8.3 Creating Mnemonics for Bit Patterns

The mnemonic table feature allows you to assign a meaningful name to a set of bit patterns, such as a bus. To create a mnemonic table, right-click in the **Setup** or **Data** tab of an `.stp` and click **Mnemonic Table Setup**. You create a mnemonic table by entering sets of bit patterns and specifying a label to represent each pattern. Once you have created a mnemonic table, assign the table to a group of signals. To assign a mnemonic table, right-click on the group, click **Bus Display Format** and select the desired mnemonic table.

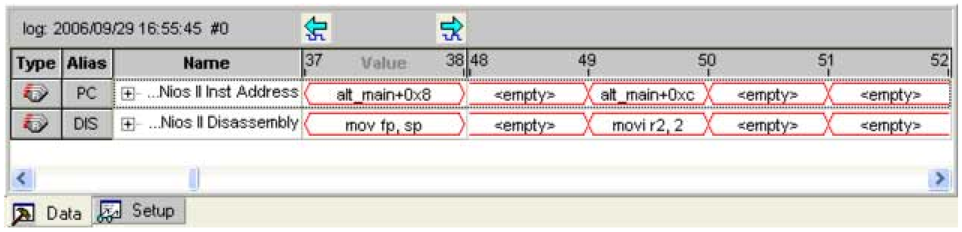
You use the labels you create in a table in different ways on the **Setup** and **Data** tabs. On the **Setup** tab, you can create basic triggers with meaningful names by right-clicking an entry in the **Trigger Conditions** column and selecting a label from the table you assigned to the signal group. On the **Data** tab, if any captured data matches a bit pattern contained in an assigned mnemonic table, the signal group data is replaced with the appropriate label, making it easy to see when expected data patterns occur.

14.8.4 Automatic Mnemonics with a Plug-In

When you use a plug-in to add signals to an `.stp`, mnemonic tables for the added signals are automatically created and assigned to the signals defined in the plug-in. To enable these mnemonic tables manually, right-click the name of the signal or signal group. On the **Bus Display Format** shortcut menu, then click the name of the mnemonic table that matches the plug-in.

As an example, the Nios II plug-in helps you to monitor signal activity for your design as the code is executed. If you set up the logic analyzer to trigger on a function name in your Nios II code based on data from an `.elf`, you can see the function name in the **Instance Address** signal group at the trigger sample, along with the corresponding disassembled code in the **Disassembly** signal group, as shown in Figure 13–52. Captured data samples around the trigger are referenced as offset addresses from the trigger function name.

Figure 212. Data Tab when the Nios II Plug-In is Used



14.8.5 Locating a Node in the Design

When you find the source of an error in your design using the Signal Tap Logic Analyzer, you can use the node locate feature to locate that signal in many of the tools found in the Intel Quartus Prime software, as well as in your design files. This lets you find the source of the problem quickly so you can modify your design to correct the flaw. To locate a signal from the Signal Tap Logic Analyzer in one of the Intel Quartus Prime software tools or your design files, right-click the signal in the `.stp`, and click **Locate in <tool name>**.

You can locate a signal from the node list with the following tools:

- Assignment Editor
- Pin Planner
- Timing Closure Floorplan
- Chip Planner
- Resource Property Editor
- Technology Map Viewer
- RTL Viewer
- Design File

14.8.6 Saving Captured Data

When you save a data capture, Signal Tap Logic Analyzer stores this data in the active `.stp` file, and the Data Log adds the capture as a log entry under the current configuration.

When analysis is set to **Auto-run mode**, the Logic Analyzer creates a separate entry in the Data Log to store the data captured each time the trigger occurred. This allows you to review the captured data for each trigger event.

The default name for a log is based time stamp when the Logic Analyzer acquired the data. As a best practice, rename the data log with a more meaningful name.

The organization of logs is hierarchical; the Logic Analyzer groups similar logs of captured data in trigger sets.

Related Links

Data Log Pane on page 350

14.8.7 Exporting Captured Data to Other File Formats

You can export captured data to the following file formats, for use with other EDA simulation tools:

- Comma Separated Values File (`.csv`)
- Table File (`.tbl`)
- Value Change Dump File (`.vcd`)
- Vector Waveform File (`.vwf`)
- Graphics format files (`.jpg`, `.bmp`)

To export the captured data from Signal Tap Logic Analyzer, on the File menu, click **Export** and specify the **File Name**, **Export Format**, and **Clock Period**.

14.8.8 Creating a Signal Tap List File

A `.stp` list file contains all the data the logic analyzer captures for a trigger event, in text format.

Each row of the list file corresponds to one captured sample in the buffer. Columns correspond to the value of each of the captured signals or signal groups for that sample. If you defined a mnemonic table for the captured data, a matching entry from the table replaces the numerical values in the list.

The `.stp` list file is especially useful when combined with a plug-in that includes instruction code disassembly. You can view the order of instruction code execution during the same time period of the trigger event.

To create a `.stp` list file in the Intel Quartus Prime software, click **File ➤ Create/ Update ➤ Create Signal Tap List File**.

Related Links

Adding Signals with a Plug-In on page 337

14.9 Other Features

The Signal Tap Logic Analyzer provides optional features not specific to a task flow. The following techniques may offer advantages in specific circumstances.

14.9.1 Creating Signal Tap File from Design Instances

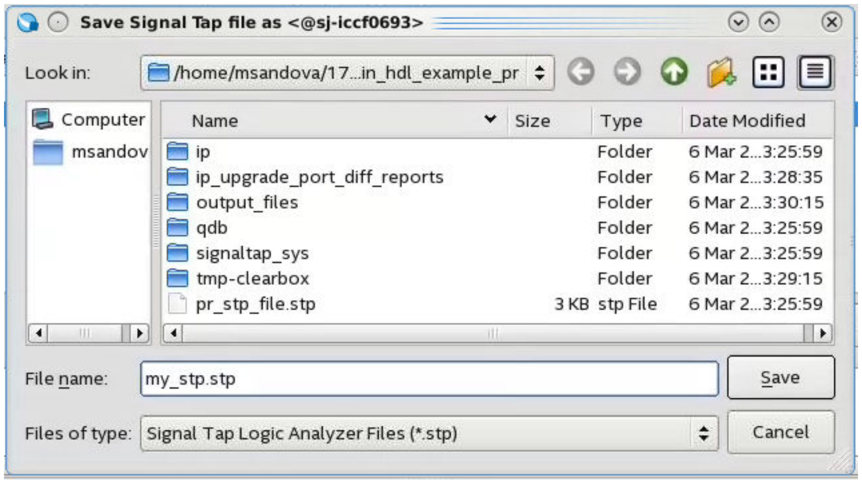
In addition to providing GUI support for generation of `.stp` files, the Intel Quartus Prime software supports generation of a Signal Tap instance from logic defined in HDL source files. This technique is helpful to modify runtime configurable trigger conditions, acquire data, and view acquired data on the data log via Signal Tap utilities.

14.9.1.1 Creating a .stp File from a Design Instance

To generate a `.stp` file from parameterized HDL instances within your design:

1. Open or create an Intel Quartus Prime project that includes one or more HDL instances of the Signal Tap logic analyzer.
2. Click **Processing ➤ Start ➤ Start Analysis & Synthesis**.
3. Click **File ➤ Create/Update ➤ Create Signal Tap File from Design Instance(s)**.
4. Specify a location for the `.stp` file that generates, and click **Save**.

Figure 213. Create Signal Tap File from Design Instances Dialog Box

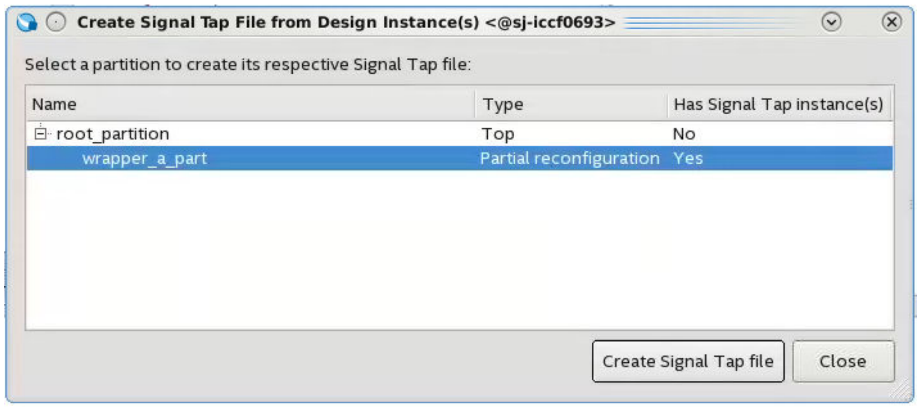

Save Signal Tap file as <@sj-iccf0693>
Look in:
/home/msandova/17...in_hdl_example_pr
Computer
msandov
Save
Cancel

Name	Size	Type	Date Modified
ip		Folder	6 Mar 2...3:25:59
ip_upgrade_port_diff_reports		Folder	6 Mar 2...3:28:35
output_files		Folder	6 Mar 2...3:30:15
qdb		Folder	6 Mar 2...3:25:59
signaltap_sys		Folder	6 Mar 2...3:25:59
tmp-clearbox		Folder	6 Mar 2...3:29:15
pr_stp_file.stp	3 KB	stp File	6 Mar 2...3:25:59

File name: my_stp.stp
Files of type: Signal Tap Logic Analyzer Files (*.stp)

Note: If your project contains partial reconfiguration partitions, the **Create Signal Tap File from Design Instance(s)** dialog box displays a tree view of the PR partitions in the project. Select a partition from the view, and click **Create Signal Tap file**. The resultant `.stp` file that generates contains all HDL instances in the corresponding PR partition. The resultant `.stp` file does not include the instances in any nested partial reconfiguration partition.

Figure 214. Selecting Partition for `.stp` File Generation

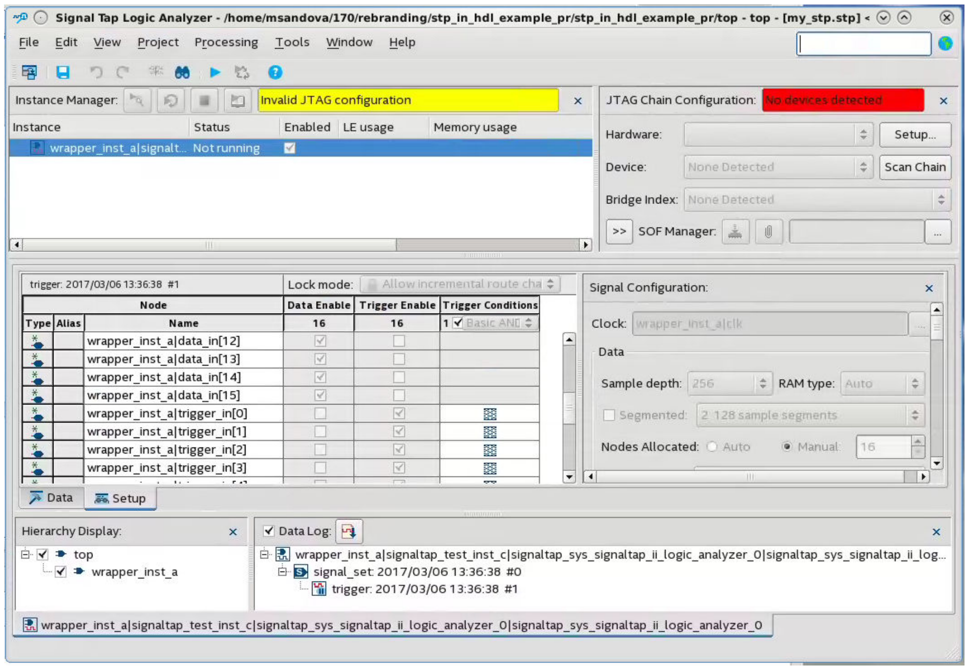

Create Signal Tap File from Design Instance(s) <@sj-iccf0693>
Create Signal Tap file
Close

Select a partition to create its respective Signal Tap file:

Name	Type	Has Signal Tap instance(s)
root_partition	Top	No
wrapper_a_part	Partial reconfiguration	Yes

After successful `.stp` file creation, the **Signal Tap Logic Analyzer** appears. All the fields are read-only, except runtime-configurable trigger conditions.

Figure 215. Generated `.stp` File



Signal Tap Logic Analyzer - /home/msandova/170/rebranding/stp_in_hdl_example_pr/stp_in_hdl_example_pr/top - top - [my_stp.stp]

Instance Manager: Invalid JTAG configuration

Instance	Status	Enabled	LE usage	Memory usage
wrapper_inst_a|signalt...	Not running	✓		

JTAG Chain Configuration: No devices detected

Hardware:
Device: None Detected
Bridge Index: None Detected
SOF Manager:

trigger: 2017/03/06 13:36:38 #1 Lock mode: Allow incremental route cha

Type	Alias	Name	Data Enable 16	Trigger Enable 16	Trigger Conditions 1 ✓ Basic AND
		wrapper_inst_a|data_in[12]	✓		
		wrapper_inst_a|data_in[13]	✓		
		wrapper_inst_a|data_in[14]	✓		
		wrapper_inst_a|data_in[15]	✓		
		wrapper_inst_a|trigger_in[0]		✓	▨
		wrapper_inst_a|trigger_in[1]		✓	▨
		wrapper_inst_a|trigger_in[2]		✓	▨
		wrapper_inst_a|trigger_in[3]		✓	▨

Signal Configuration:
Clock: wrapper_inst_a|clk
Data
Sample depth: 256 RAM type: Auto
Segmented: 2 128 sample segments
Nodes Allocated: Auto [✓ Manual] 16

Hierarchy Display:
top
wrapper_inst_a

Data Log:
wrapper_inst_a|signaltap_test_inst_c|signaltap_sys_signaltap_ii_logic_analyzer_0|signaltap_sys_signaltap_ii_log...
signal_set: 2017/03/06 13:36:38 #0
trigger: 2017/03/06 13:36:38 #1

wrapper_inst_a|signaltap_test_inst_c|signaltap_sys_signaltap_ii_logic_analyzer_0|signaltap_sys_signaltap_ii_logic_analyzer_0

Related Links

- Create Signal Tap File from Design Instances
  In *Intel Quartus Prime Help*
- Custom Trigger HDL Object on page 358

14.9.2 Using the Signal Tap MATLAB MEX Function to Capture Data

When you use MATLAB for DSP design, you can acquire data from the Signal Tap Logic Analyzer directly into a matrix in the MATLAB environment by calling the MATLAB MEX function `alt_signaltap_run`, built into the Intel Quartus Prime software. If you use the MATLAB MEX function in a loop, you can perform as many acquisitions in the same amount of time as you can when using Signal Tap in the Intel Quartus Prime software environment.

Note: The Signal Tap MATLAB MEX function is available in the Windows* version and Linux version of the Intel Quartus Prime software. This function is compatible with MATLAB Release 14 Original Release Version 7 and later.

To set up the Intel Quartus Prime software and the MATLAB environment to perform Signal Tap acquisitions:

1. In the Intel Quartus Prime software, create an `.stp` file.
2. In the node list in the **Data** tab of the Signal Tap Logic Analyzer Editor, organize the signals and groups of signals into the order in which you want them to appear in the MATLAB matrix.

   Each column of the imported matrix represents a single Signal Tap acquisition sample, while each row represents a signal or group of signals in the order you defined in the **Data** tab.

Note: Signal groups that the Signal Tap Logic Analyzer acquires and transfers into the MATLAB MEX function have a width limit of 32 signals. To use the MATLAB MEX function with a bus or signal group that contains more than 32 signals, split the group into smaller groups that do not exceed the limit.

3. Save the `.stp` file and compile your design. Program your device and run the Signal Tap Logic Analyzer to ensure your trigger conditions and signal acquisition work correctly.
4. In the MATLAB environment, add the Intel Quartus Prime binary directory to your path with the following command:

```
addpath <Quartus install directory>\win
```

   You can view the help file for the MEX function by entering the following command in MATLAB without any operators:

   `alt_signaltap_run`

5. Use the MATLAB MEX function to open the JTAG connection to the device and run the Signal Tap Logic Analyzer to acquire data. When you finish acquiring data, close the JTAG connection.

   To open the JTAG connection and begin acquiring captured data directly into a MATLAB matrix called `stp`, use the following command:

```
stp = alt_signaltap_run \
('<stp filename>'[,('signed'|'unsigned')[,'<instance names>'[, \
'<signalset name>'[,'<trigger name>']]]]);
```

   When capturing data, you must assign a filename, for example, *<stp filename>* as a requirement of the MATLAB MEX function. Other MATLAB MEX function options are described in the table:

Table 121. Signal Tap MATLAB MEX Function Options

Option	Usage	Description
`signed` `unsigned`	`'signed'` `'unsigned'`	The **signed** option turns signal group data into 32-bit two's-complement signed integers. The MSB of the group as defined in the Signal Tap **Data** tab is the sign bit. The **unsigned** option keeps the data as an unsigned integer. The default is **signed**.
<instance name>	`'auto_signaltap_0'`	Specify a Signal Tap instance if more than one instance is defined. The default is the first instance in the `.stp`, `auto_signaltap_0`.
<signal set name> *<trigger name>*	`'my_signalset'` `'my_trigger'`	Specify the signal set and trigger from the Signal Tap data log if multiple configurations are present in the `.stp`. The default is the active signal set and trigger in the file.

During data acquisition, you can enable or disable verbose mode to see the status of the logic analyzer. To enable or disable verbose mode, use the following commands:

```
alt_signaltap_run('VERBOSE_ON');-alt_signaltap_run('VERBOSE_OFF');
```

When you finish acquiring data, close the JTAG connection with the following command:

```
alt_signaltap_run('END_CONNECTION');
```

For more information about the use of MATLAB MEX functions in MATLAB, refer to the MATLAB Help.

14.9.3 Using Signal Tap in a Lab Environment

You can install a stand-alone version of the Signal Tap Logic Analyzer. This version is particularly useful in a lab environment in which you do not have a workstation that meets the requirements for a complete Intel Quartus Prime installation, or if you do not have a license for a full installation of the Intel Quartus Prime software. The standalone version of the Signal Tap Logic Analyzer is included with and requires the Intel Quartus Prime stand-alone Programmer which is available from the Downloads page of the Altera website.

14.9.4 Remote Debugging Using the Signal Tap Logic Analyzer

14.9.4.1 Debugging Using a Local PC and an SoC

You can use the System Console with Signal Tap Logic Analyzer to remote debug your Intel FPGA SoC. This method requires one local PC, an existing TCP/IP connection, a programming device at the remote location, and an Intel FPGA SoC.

Related Links

Remote Hardware Debugging over TCP/IP

14.9.4.2 Debugging Using a Local PC and a Remote PC

You can use the Signal Tap Logic Analyzer to debug a design that is running on a device attached to a PC in a remote location.

To perform a remote debugging session, you must have the following setup:

- The Intel Quartus Prime software installed on the local PC
- Stand-alone Signal Tap Logic Analyzer or the full version of the Intel Quartus Prime software installed on the remote PC
- Programming hardware connected to the device on the PCB at the remote location
- TCP/IP protocol connection

14.9.4.2.1 Equipment Setup

1. On the PC in the remote location, install the standalone version of the Signal Tap Logic Analyzer, included in the Intel Quartus Prime stand-alone Programmer, or the full version of the Intel Quartus Prime software.
2. Connect the remote computer to Intel programming hardware, such as the or Intel FPGA Download Cable.
3. On the local PC, install the full version of the Intel Quartus Prime software.
4. Connect the local PC to the remote PC across a LAN with the TCP/IP protocol.

14.9.5 Using the Signal Tap Logic Analyzer in Devices with Configuration Bitstream Security

Certain device families support bitstream decryption during configuration using an on-device AES decryption engine. You can still use the Signal Tap Logic Analyzer to analyze functional data within the FPGA. However, note that JTAG configuration is not possible after the security key has been programmed into the device.

Intel FPGA recommends that you use an unencrypted bitstream during the prototype and debugging phases of the design. Using an unencrypted bitstream allows you to generate new programming files and reconfigure the device over the JTAG connection during the debugging cycle.

If you must use the Signal Tap Logic Analyzer with an encrypted bitstream, first configure the device with an encrypted configuration file using Passive Serial (PS), Fast Passive Parallel (FPP), or Active Serial (AS) configuration modes. The design must contain at least one instance of the Signal Tap Logic Analyzer. After the FPGA is configured with a Signal Tap Logic Analyzer instance in the design, when you open the Signal Tap Logic Analyzer in the Intel Quartus Prime software, you then scan the chain and are ready to acquire data with the JTAG connection.

14.9.6 Monitor FPGA Resources Used by the Signal Tap Logic Analyzer

The Signal Tap Logic Analyzer has a built-in resource estimator that calculates the logic resources and amount of memory that each logic analyzer instance uses. Furthermore, because the most demanding on-chip resource for the logic analyzer is memory usage, the resource estimator reports the ratio of total RAM usage in your design to the total amount of RAM available, given the results of the last compilation. The resource estimator provides a warning if a potential for a “no-fit” occurs.

You can see resource usage (by instance and total) in the columns of the **Instance Manager** pane of the Signal Tap Logic Analyzer Editor. Use this feature when you know that your design is running low on resources.

The logic element value that the resource usage estimator reports may vary by as much as 10% from the actual resource usage.

14.10 Design Example: Using Signal Tap Logic Analyzers

The system in this example contains many components, including a Nios processor, a direct memory access (DMA) controller, on-chip memory, and an interface to external SDRAM memory. After you press a button, the processor initiates a DMA transfer, which you analyze using the Signal Tap Logic Analyzer. In this example, the Nios processor executes a simple C program from on-chip memory and waits for you to press a button.

Related Links

AN 446: Debugging Nios II Systems with the Signal Tap Embedded Logic Analyzer application note

14.11 Custom Triggering Flow Application Examples

The custom triggering flow in the Signal Tap Logic Analyzer is most useful for organizing a number of triggering conditions and for precise control over the acquisition buffer. This section provides two application examples for defining a custom triggering flow within the Signal Tap Logic Analyzer. Both examples can be easily copied and pasted directly into the state machine description box by using the state display mode **All states in one window**.

Related Links

On-chip Debugging Design Examples website

14.11.1 Design Example 1: Specifying a Custom Trigger Position

Actions to the acquisition buffer can accept an optional post-count argument. This post-count argument enables you to define a custom triggering position for each segment in the acquisition buffer.

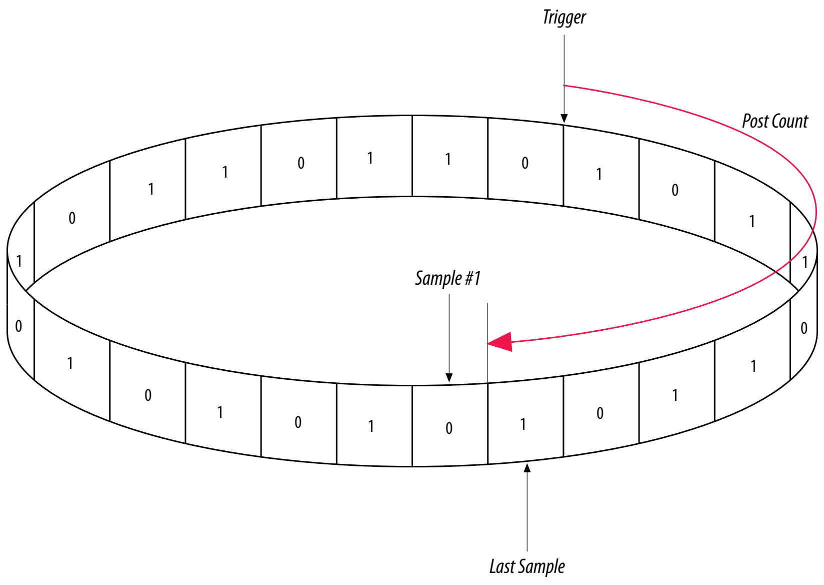
The example shows how to apply a trigger position to all segments in the acquisition buffer. The example describes a triggering flow for an acquisition buffer split into four segments. If each acquisition segment is 64 samples in depth, the trigger position for each buffer will be at sample #34. The acquisition stops after all four segments are filled once.

```
if (c1 == 3 && condition1)
    trigger 30;
else if ( condition1 )
begin
    segment_trigger 30;
    increment c1;
end
```

Each segment acts as a non-segmented buffer that continuously updates the memory contents with the signal values.

The **Data** tab displays the last acquisition before stopping the buffer as the last sample number in the affected segment. The trigger position in the affected segment is then defined by `N – post count fill`, where N is the number of samples per segment.

Figure 216. Specifying a Custom Trigger Position



14.11.2 Design Example 2: Trigger When triggercond1 Occurs Ten Times between triggercond2 and triggercond3

The custom trigger flow description is often useful to count a sequence of events before triggering the acquisition buffer. The example shows such a sample flow. This example uses three basic triggering conditions configured in the Signal Tap **Setup** tab.

This example triggers the acquisition buffer when `condition1` occurs after `condition3` and occurs ten times prior to `condition3`. If `condition3` occurs prior to ten repetitions of `condition1`, the state machine transitions to a permanent wait state.

```
state ST1:
if ( condition2  )
begin
   reset c1;
   goto ST2;
end
State ST2 :
if ( condition1 )
    increment c1;
else if (condition3 && c1 < 10)
    goto ST3;
else if ( condition3 && c1 >= 10)
    trigger;
ST3:
goto ST3;
```

14.12 Signal Tap Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Intel Quartus Prime Command-Line and Tcl API Help browser. To run the Help browser, type the following at the command prompt:

```
quartus_sh --qhelp
```

Related Links

Tcl Scripting
In *Intel Quartus Prime Standard Edition Handbook Volume 2*

14.12.1 Signal Tap Tcl Commands

The `quartus_stp` executable supports a Tcl interface that allows you to capture data without running the Intel Quartus Prime GUI. You cannot execute Signal Tap Tcl commands from within the Tcl console in the Intel Quartus Prime software. You must run them from the command-line with the `quartus_stp` executable. To execute a Tcl file that has Signal Tap Logic Analyzer Tcl commands, use the following command:

```
quartus_stp -t <Tcl file>
```

Example 38. Continuously capturing data

This excerpt shows commands you can use to continuously capture data. Once the capture meets trigger condition e, the data is captured and stored in the data log.

```
#  Open Signal Tap session
open_session -name stp1.stp

###  Start acquisition of instances auto_signaltap_0 and
###  auto_signaltap_1 at the same time

#  Calling run_multiple_end will start all instances
run_multiple_start

run -instance auto_signaltap_0 -signal_set signal_set_1 -trigger \
trigger_1 -data_log log_1 -timeout 5
run -instance auto_signaltap_1 -signal_set signal_set_1 -trigger \
trigger_1 -data_log log_1 -timeout 5

run_multiple_end

# Close Signal Tap session
close_session
```

Related Links

::quartus::stp
In *Intel Quartus Prime Help*

14.12.2 Signal Tap Command-Line Options

To compile your design with the Signal Tap Logic Analyzer using the command prompt, use the `quartus_stp` command. You can use the following options with the `quartus_stp` executable:

Table 122. `quartus_stp` Command-Line Options

Option	Usage	Description
`--stp_file` *<stp_filename>*	Mandatory	Specifies the name of the `.stp` file.
`--enable`	Optional	Creates assignments to the specified `.stp` in the `.qsf` and changes ENABLE_SIGNALTAP to ON. Includes Signal Tap Logic Analyzer in the next compilation. If no `.stp` is specified in the `.qsf`, the `--stp_file` option must be used. If omitted, the compiler uses the current value of ENABLE_SIGNALTAP in the `.qsf file` .
`--disable`	Optional	Removes the `.stp` reference from the `.qsf` and changes ENABLE_SIGNALTAP to OFF. The Signal Tap Logic Analyzer is removed from the design database the next time you compile your design. If the `--disable` option is omitted, the current value of ENABLE_SIGNALTAP in the `.qsf` is used.
`--create_signaltap_hdl_file`	Optional	Creates an `.stp` representing the Signal Tap instance. You must use the `--stp_file` option to create an `.stp`. Equivalent to the **Create Signal Tap File from Design Instance(s)** command in the Intel Quartus Prime software.

The first example illustrates how to compile a design with the Signal Tap Logic Analyzer at the command line.

```
quartus_stp filtref --stp_file stp1.stp --enable
quartus_map filtref --source=filtref.bdf --family=CYCLONE
quartus_fit filtref --part=EP1C12Q240C6 --fmax=80MHz --tsu=8ns
quartus_asm filtref
```

The `quartus_stp --stp_file stp1.stp --enable` command creates the QSF variable and instructs the Intel Quartus Prime software to compile the `stp1.stp` file with your design. The `--enable` option must be applied for the Signal Tap Logic Analyzer to compile into your design.

The example below shows how to create a new `.stp` after building the Signal Tap Logic Analyzer instance with the IP Catalog.

```
quartus_stp filtref --create_signaltap_hdl_file --stp_file stp1.stp
```

Related Links

Command-Line Scripting
In *Intel Quartus Prime Standard Edition Handbook Volume 2*

14.13 Document Revision History

Table 123. Document Revision History

Date	Version	Changes Made
2017.11.06	17.1.0	• Clarified information about the Data Log Pane. • Updated Figure: Data Log and renamed to Simple Data Log. • Added Figure: Accessing the Advanced Trigger Condition Tab.
2017.05.08	17.0.0	• Added: Open Standalone Signal Tap Logic Analyzer GUI. • Updated figures on Create Signal Tap File from Design Instance(s).
2016.10.31	16.1.0	• Added: Create Signal Tap File from Design Instance(s). • Removed reference to unsupported Talkback feature.
2016.05.03	16.0.0	• Added: Specifying the Pipeline Factor • Added: Comparison Trigger Conditions
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
2015.05.04	15.0.0	Added content for Floating Point Display Format in table: Signal Tap Logic Analyzer Features and Benefits.
2014.12.15	14.1.0	Updated location of Fitter Settings, Analysis & Synthesis Settings, and Physical Synthesis Optimizations to Compiler Settings.
December 2014	14.1.0	• Added MAX 10 as supported device. • Removed Full Incremental Compilation setting and Post-Fit (Strict) netlist type setting information. • Removed outdated GUI images from "Using Incremental Compilation with the Signal Tap Logic Analyzer" section.
June 2014	14.0.0	• DITA conversion. • Replaced MegaWizard Plug-In Manager and Megafunction content with IP Catalog and parameter editor content. • Added flows for custom trigger HDL object, Incremental Route with Rapid Recompile, and nested groups with Basic OR. • GUI changes: toolbar, drag to zoom, disable/enable instance, trigger log time-stamping.
November 2013	13.1.0	Removed HardCopy material. Added section on using cross-triggering with DS-5 tool and added link to white paper 01198. Added section on remote debugging an Altera SoC and added link to application note 693. Updated support for MEX function.
May 2013	13.0.0	• Added recommendation to use the state-based flow for segmented buffers with separate trigger conditions, information about Basic OR trigger condition, and hard processor system (HPS) external triggers. • Updated “Segmented Buffer” on page 13-17, Conditional Mode on page 13-21, Creating Basic Trigger Conditions on page 13-16, and Using External Triggers on page 13-48.
June 2012	12.0.0	Updated Figure 13–5 on page 13–16 and “Adding Signals to the Signal Tap File” on page 13–10.
November 2011	11.0.1	Template update. Minor editorial updates.
May 2011	11.0.0	Updated the requirement for the standalone Signal Tap software.
December 2010	10.0.1	Changed to new document template.

continued...

Date	Version	Changes Made
July 2010	10.0.0	• Add new acquisition buffer content to the "View, Analyze, and Use Captured Data" section. • Added script sample for generating hexadecimal CRC values in programmed devices. • Created cross references to Intel Quartus Prime Help for duplicated procedural content.
November 2009	9.1.0	No change to content.
March 2009	9.0.0	• Updated Table 13–1 • Updated "Using Incremental Compilation with the Signal Tap Logic Analyzer" on page 13–45 • Added new Figure 13–33 • Made minor editorial updates
November 2008	8.1.0	Updated for the Intel Quartus Prime software version 8.1 release: • Added new section "Using the Storage Qualifier Feature" on page 14–25 • Added description of `start_store` and `stop_store` commands in section "Trigger Condition Flow Control" on page 14–36 • Added new section "Runtime Reconfigurable Options" on page 14–63
May 2008	8.0.0	Updated for the Intel Quartus Prime software version 8.0: • Added "Debugging Finite State machines" on page 14-24 • Documented various GUI usability enhancements, including improvements to the resource estimator, the bus find feature, and the dynamic display updates to the counter and flag resources in the State-based trigger flow control tab • Added "Capturing Data Using Segmented Buffers" on page 14–16 • Added hyperlinks to referenced documents throughout the chapter • Minor editorial updates

Related Links

Documentation Archive

For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

16 Debugging Single Event Upset Using the Fault Injection Debugger

You can detect and debug single event upset (SEU) using the Fault Injection Debugger in the Intel Quartus Prime software. Use the debugger with the Intel FPGA Fault Injection IP core to inject errors into the configuration RAM (CRAM) of an Intel FPGA device.

The injected error simulates the soft errors that can occur during normal operation due to SEUs. Since SEUs are rare events, and therefore difficult to test, you can use the Fault Injection Debugger to induce intentional errors in the FPGA to test the system's response to these errors.

The Fault Injection Debugger is available for Intel Arria 10 and Stratix V family devices. For assistance with support for Arria V or Cyclone V family devices, file a service request using your myAltera account.

The Fault Injection Debugger provides the following benefits:

- Allows you to evaluate system response for mitigating single event functional interrupts (SEFI).
- Allows you to perform SEFI characterization, eliminating the need for entire system beam testing. Instead, you can limit the beam testing to failures in time (FIT)/Mb measurement at the device level.
- Scale FIT rates according to the SEFI characterization that is relevant to your design architecture. You can randomly distribute fault injections throughout the entire device, or constrain them to specific functional areas to speed up testing.
- Optimize your design to reduce SEU-caused disruption.

Related Links

- myAltera Log In
- Single Event Upsets
- AN 737: SEU Detection and Recovery in Intel Arria 10 Devices
- Understanding Single Event Functional Interrupts in FPGA Designs White Paper

16.1 Single Event Upset Mitigation

Integrated circuits and programmable logic devices such as FPGAs are susceptible to SEUs. SEUs are random, nondestructive events, caused by two major sources: alpha particles and neutrons from cosmic rays. Radiation can cause either the logic register, embedded memory bit, or a configuration RAM (CRAM) bit to flip its state, thus leading to unexpected device operation.

Intel Arria 10, Arria V, Cyclone V, Stratix V and newer devices have the following CRAM capabilities:

- Error Detection Cyclical Redundance Checking (EDCRC)
- Automatic correction of an upset CRAM (scrubbing)
- Ability to create an upset CRAM condition (fault injection)

For more information about SEU mitigation in Intel FPGA devices, refer to the *SEU Mitigation* chapter in the respective device handbook.

Related Links

- SEU Mitigation in Intel Arria 10 Devices: Hierarchy Tagging Online Course
- Mitigating Single Event Upsets in Intel Arria 10 Devices Online Course

16.2 Hardware and Software Requirements

The following hardware and software is required to use the Fault Injection Debugger:

- Intel Quartus Prime software version 14.0 or later.
- FEATURE line in your Intel FPGA license that enables the Fault Injection IP core. For more information, contact your local Intel FPGA sales representative.
- Download cable (Intel FPGA Download Cable, Intel FPGA Download Cable II, Intel FPGA Ethernet Cable, or Intel FPGA Ethernet Cable II).
- Intel FPGA development kit or user designed board with a JTAG connection to the device under test.
- (Optional) FEATURE line in your Intel FPGA license that enables the Advanced SEU Detection IP core.

Related Links

Altera Website: Contact Us

16.3 Using the Fault Injection Debugger and Fault Injection IP Core

The Fault Injection Debugger works together with the Fault Injection IP core. First, you instantiate the IP core in your design, compile, and download the resulting configuration file into your device. Then, you run the Fault Injection Debugger from within the Intel Quartus Prime software or from the command line to simulate soft errors.

- The Fault Injection Debugger allows you to operate fault injection experiments interactively or by batch commands, and allows you to specify the logical areas in your design for fault injections.
- The command-line interface is useful for running the debugger via a script.

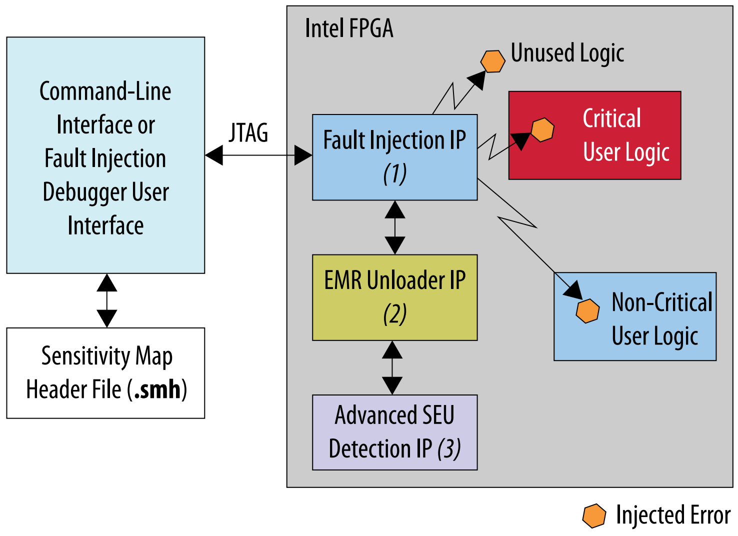
The Fault Injection Debugger communicates with the Fault Injection IP core via the JTAG interface. The Fault Injection IP accepts commands from the JTAG interface and reports status back through the JTAG interface.

Note: The Fault Injection IP core is implemented in soft logic in your device; therefore, you must account for this logic usage in your design. One methodology is to characterize your design’s response to SEU in the lab and then omit the IP core from your final deployed design.

You use the Fault Injection IP core with the following IP cores:

- The Error Message Register (EMR) Unloader IP core, which reads and stores data from the hardened error detection circuitry in Intel FPGA devices.
- (Optional) The Advanced SEU Detection (ASD) IP core, which compares single-bit error locations to a sensitivity map during device operation to determine whether a soft error affects it.

Figure 217. Fault Injection Debugger Overview Block Diagram


Command-Line Interface or Fault Injection Debugger User Interface
JTAG
Sensitivity Map Header File (.smh)
Intel FPGA
Unused Logic
Fault Injection IP (1)
Critical User Logic
EMR Unloader IP (2)
Non-Critical User Logic
Advanced SEU Detection IP (3)
Injected Error

Notes:
1. The fault Injection IP flips the bits of the targeted logic.
2. The Fault Injection Debugger and Advanced SEU Detection IP use the same EMR Unloader instance.
3. The Advanced SEU Detection IP core is optional.

Related Links

- About SMH Files on page 408
- AN 539: Test Methodology or Error Detection and Recovery using CRC in Intel FPGA Devices
- Instantiating the Intel FPGA Fault Injection IP Core on page 405
- About the EMR Unloader IP Core on page 406
- About the Advanced SEU Detection IP Core on page 407

16.3.1 Instantiating the Intel FPGA Fault Injection IP Core

The Fault Injection IP core does not require you to set any parameters. To use the IP core, create a new IP instance, include it in your Platform Designer (Standard) system, and connect the signals as appropriate.

Note: You must use the Fault Injection IP core with the Error Message Register (EMR) Unloader IP core.

The Fault Injection and the EMR Unloader IP cores are available in Platform Designer (Standard) and the IP Catalog. Optionally, you can instantiate them directly into your RTL design, using Verilog HDL, SystemVerilog, or VHDL.

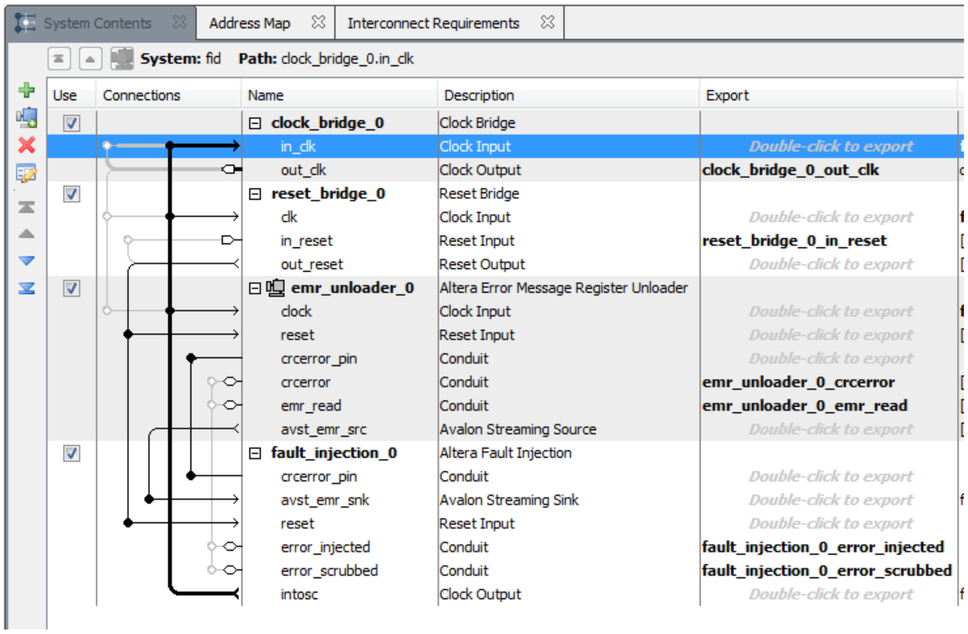
Related Links

Intel FPGA Fault Injection IP Core User Guide

16.3.1.1 About the EMR Unloader IP Core

The EMR Unloader IP core provides an interface to the EMR, which is updated continuously by the device’s EDCRC that checks the device's CRAM bits CRC for soft errors.

Figure 218. Example Platform Designer (Standard) System Including the Fault Injection IP Core and EMR Unloader IP Core

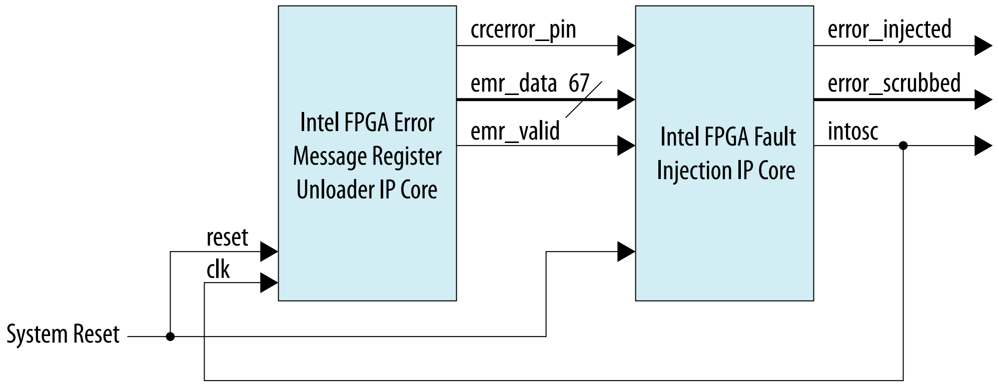


System Contents | Address Map | Interconnect Requirements

System: fid **Path:** clock_bridge_0.in_clk

Use	Connections	Name	Description	Export
✓		**clock_bridge_0**	Clock Bridge	
		in_clk	Clock Input	*Double-click to export*
		out_clk	Clock Output	**clock_bridge_0_out_clk**
✓		**reset_bridge_0**	Reset Bridge	
		clk	Clock Input	*Double-click to export*
		in_reset	Reset Input	**reset_bridge_0_in_reset**
		out_reset	Reset Output	*Double-click to export*
✓		**emr_unloader_0**	Altera Error Message Register Unloader	
		clock	Clock Input	*Double-click to export*
		reset	Reset Input	*Double-click to export*
		crcerror_pin	Conduit	*Double-click to export*
		crcerror	Conduit	**emr_unloader_0_crcerror**
		emr_read	Conduit	**emr_unloader_0_emr_read**
		avst_emr_src	Avalon Streaming Source	*Double-click to export*
✓		**fault_injection_0**	Altera Fault Injection	
		crcerror_pin	Conduit	*Double-click to export*
		avst_emr_snk	Avalon Streaming Sink	*Double-click to export*
		reset	Reset Input	*Double-click to export*
		error_injected	Conduit	**fault_injection_0_error_injected**
		error_scrubbed	Conduit	**fault_injection_0_error_scrubbed**
		intosc	Clock Output	*Double-click to export*

Figure 219. Example Intel FPGA Fault Injection IP Core and EMR Unloader IP Core Block Diagram



Related Links

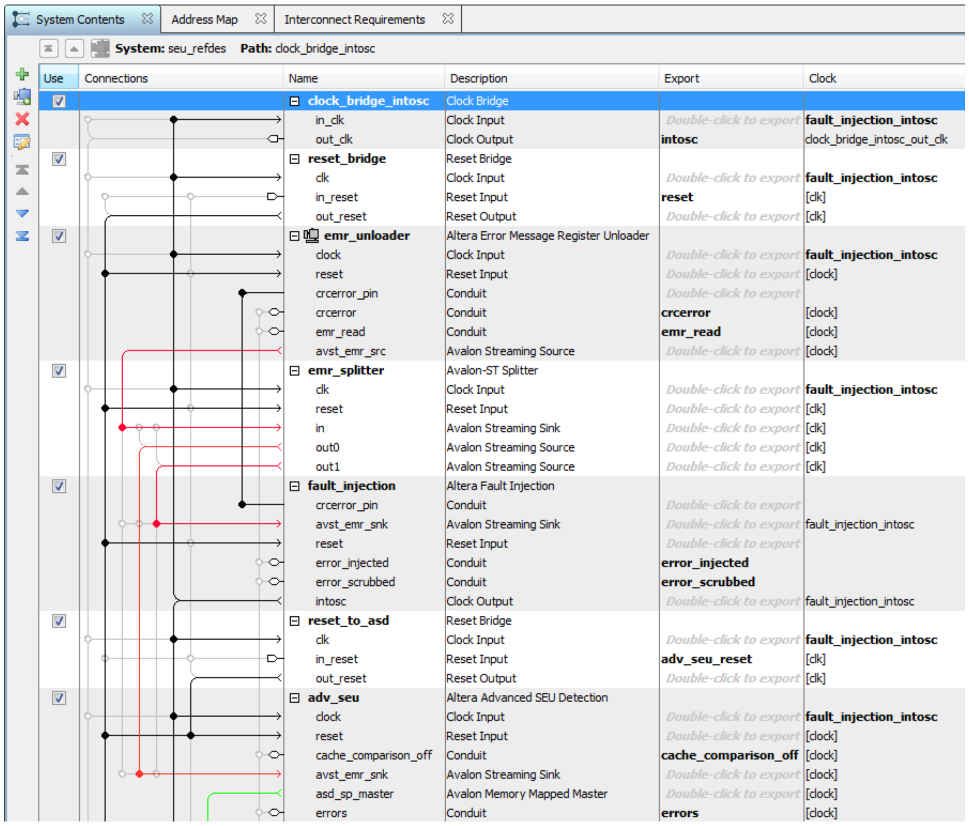
Intel FPGA Error Message Unloader IP Core User Guide

16.3.1.2 About the Advanced SEU Detection IP Core

Use the Advanced SEU Detection (ASD) IP core when SEU tolerance is a design concern.

You must use the EMR Unloader IP core with the ASD IP core. Therefore, if you use the ASD IP and the Fault Injection IP in the same design, they must share the EMR Unloader output via an Avalon-ST splitter component. The following figure shows a Platform Designer (Standard) system in which an Avalon-ST splitter distributes the EMR contents to the ASD and Fault Injection IP cores.

Figure 220. Using the ASD and Fault Injection IP in the Same Platform Designer (Standard) System



System: seu_refdes **Path:** clock_bridge_intosc

Use	Name	Description	Export	Clock
✓	**clock_bridge_intosc**	Clock Bridge		
	in_clk	Clock Input	Double-click to export	**fault_injection_intosc**
	out_clk	Clock Output	**intosc**	clock_bridge_intosc_out_clk
✓	**reset_bridge**	Reset Bridge		
	clk	Clock Input	Double-click to export	**fault_injection_intosc**
	in_reset	Reset Input	**reset**	[clk]
	out_reset	Reset Output	Double-click to export	[clk]
✓	**emr_unloader**	Altera Error Message Register Unloader		
	clock	Clock Input	Double-click to export	**fault_injection_intosc**
	reset	Reset Input	Double-click to export	[clock]
	crcerror_pin	Conduit	Double-click to export	
	crcerror	Conduit	**crcerror**	[clock]
	emr_read	Conduit	**emr_read**	[clock]
	avst_emr_src	Avalon Streaming Source	Double-click to export	[clock]
✓	**emr_splitter**	Avalon-ST Splitter		
	clk	Clock Input	Double-click to export	**fault_injection_intosc**
	reset	Reset Input	Double-click to export	[clk]
	in	Avalon Streaming Sink	Double-click to export	[clk]
	out0	Avalon Streaming Source	Double-click to export	[clk]
	out1	Avalon Streaming Source	Double-click to export	[clk]
✓	**fault_injection**	Altera Fault Injection		
	crcerror_pin	Conduit	Double-click to export	
	avst_emr_snk	Avalon Streaming Sink	Double-click to export	fault_injection_intosc
	reset	Reset Input	Double-click to export	
	error_injected	Conduit	**error_injected**	
	error_scrubbed	Conduit	**error_scrubbed**	
	intosc	Clock Output	Double-click to export	fault_injection_intosc
✓	**reset_to_asd**	Reset Bridge		
	clk	Clock Input	Double-click to export	**fault_injection_intosc**
	in_reset	Reset Input	**adv_seu_reset**	[clk]
	out_reset	Reset Output	Double-click to export	[clk]
✓	**adv_seu**	Altera Advanced SEU Detection		
	clock	Clock Input	Double-click to export	**fault_injection_intosc**
	reset	Reset Input	Double-click to export	[clock]
	cache_comparison_off	Conduit	**cache_comparison_off**	[clock]
	avst_emr_snk	Avalon Streaming Sink	Double-click to export	[clock]
	asd_sp_master	Avalon Memory Mapped Master	Double-click to export	[clock]
	errors	Conduit	**errors**	[clock]

Related Links

Intel FPGA Advanced SEU Detection IP Core User Guide

16.3.2 Defining Fault Injection Areas

You can define specific regions of the FPGA for fault injection using a Sensitivity Map Header (**.smh**) file.

The SMH file stores the coordinates of the device CRAM bits, their assigned region (ASD Region), and criticality. During the design process you use hierarchy tagging to create the region. Then, during compilation, the Intel Quartus Prime Assembler generates the SMH file. The Fault Injection Debugger limits error injections to specific device regions you define in the SMH file.

16.3.2.1 Performing Hierarchy Tagging

You define the FPGA regions for testing by assigning an ASD Region to the location. You can specify an ASD Region value for any portion of your design hierarchy using the Design Partitions Window.

1. Choose **Assignments ➤ Design Partitions Window**.
2. Right-click anywhere in the header row and turn on **ASD Region** to display the **ASD Region** column (if it is not already displayed).
3. Enter a value from 0 to 16 for any partition to assign it to a specific ASD Region.
    — ASD region 0 is reserved to unused portions of the device. You can assign a partition to this region to specify it as non-critical..
    — ASD region 1 is the default region. All used portions of the device are assigned to this region unless you explicitly change the ASD Region assignment.

16.3.2.2 About SMH Files

The SMH file contains the following information:

- If you are not using hierarchy tagging (i.e., the design has no explicit ASD Region assignments in the design hierarchy), the SMH file lists every CRAM bit and indicates whether it is sensitive for the design.
- If you have performed hierarchy tagging and changed default ASD Region assignments, the SMH file lists every CRAM bit and it's assigned ASD region.

The Fault Injection Debugger can limit injections to one or more specified regions.

Note: To direct the Assembler to generate an SMH file:

- Choose **Assignments ➤ Device ➤ Device and Pin Options ➤ Error Detection CRC**.
- Turn on the **Generate SEU sensitivity map file (.smh)** option.

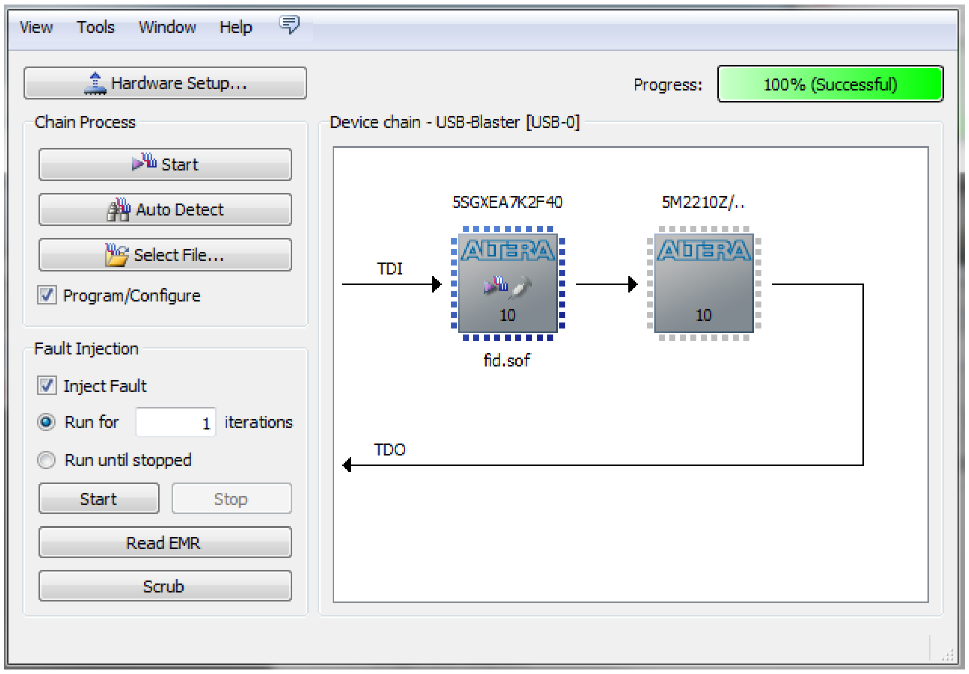
16.3.3 Using the Fault Injection Debugger

To use the Fault Injection Debugger, you connect to your device via the JTAG interface. Then, configure the device and perform fault injection.

To launch the Fault Injection Debugger, choose **Tools ➤ Fault Injection Debugger** in the Intel Quartus Prime software.

Note: Configuring or programming the device is similar to the procedure used for the Programmer or Signal Tap Logic Analyzer.

Figure 221. Fault Injection Debugger



To configure your JTAG chain:

1. Click **Hardware Setup**. The tool displays the programming hardware connected to your computer.
2. Select the programming hardware you wish to use.
3. Click **Close**.
4. Click **Auto Detect**, which populates the device chain with the programmable devices found in the JTAG chain.

Related Links

Targeted Fault Injection Feature on page 415

16.3.3.1 Configuring Your Device and the Fault Injection Debugger

The Fault Injection Debugger uses a **.sof** and (optionally) a Sensitivity Map Header (**.smh**) file.

The Software Object File (**.sof**) configures the FPGA. The **.smh** file defines the sensitivity of the CRAM bits in the device. If you do not provide an **.smh** file, the Fault Injection Debugger injects faults randomly throughout the CRAM bits.

To specify a **.sof**:

1. Select the FPGA you wish to configure in the **Device chain** box.
2. Click **Select File**.
3. Navigate to the **.sof** and click **OK**. The Fault Injection Debugger reads the **.sof**.
4. (Optional) Select the SMH file.

If you do not specify an SMH file, the Fault Injection Debugger injects faults randomly across the entire device. If you specify an SMH file, you can restrict injections to the used areas of your device.

a. Right-click the device in the **Device chain** box and then click **Select SMH File**.

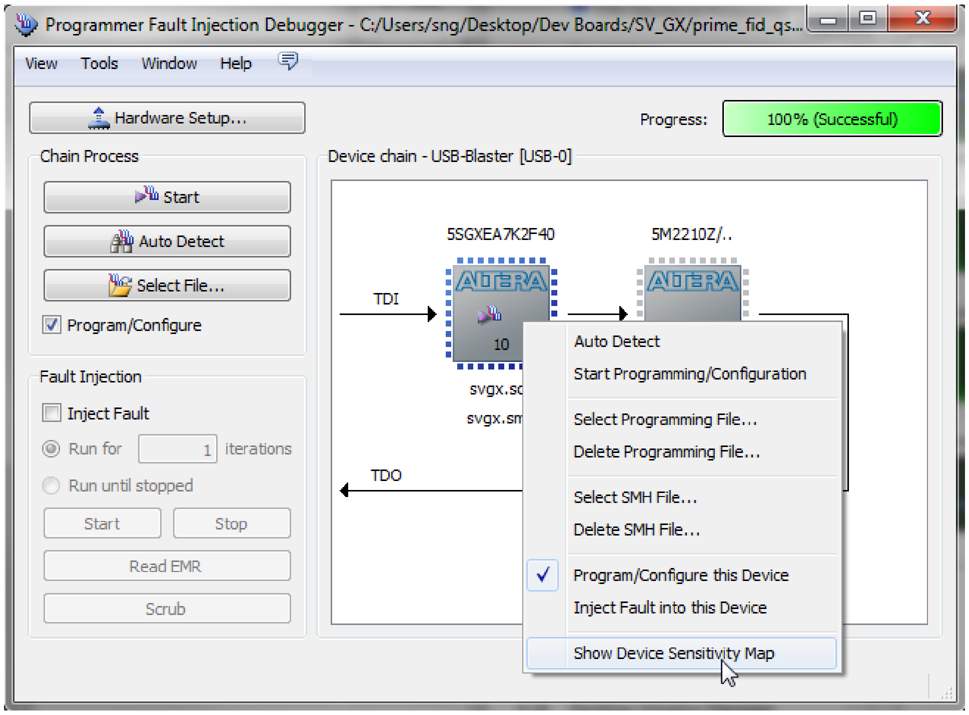
b. Select your SMH file.

c. Click **OK**.

5. Turn on **Program/Configure**.
6. Click **Start**.

The Fault Injection Debugger configures the device using the **.sof**.

Figure 222. Context Menu for Selecting the SMH File



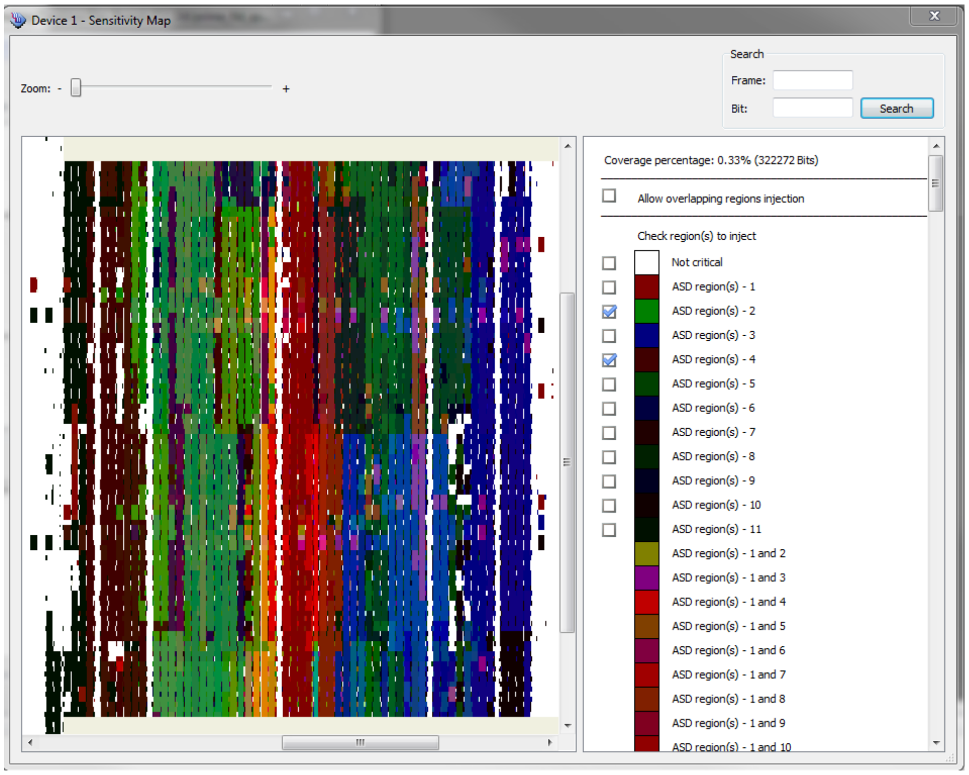
16.3.3.2 Constraining Regions for Fault Injection

After loading an SMH file, you can direct the Fault Injection Debugger to operate on only specific ASD regions.

To specify the ASD region(s) in which to inject faults:

1. Right-click the FPGA in the **Device chain** box, and click **Show Device Sensitivity Map**.
2. Select the ASD region(s) for fault injection.

Figure 223. Device Sensitivity Map Viewer



Coverage percentage: 0.33% (322272 Bits)

☐ Allow overlapping regions injection

Check region(s) to inject

☐	⬜	Not critical
☐	🟥	ASD region(s) - 1
☑	🟩	ASD region(s) - 2
☐	🟦	ASD region(s) - 3
☑	🟫	ASD region(s) - 4
☐	🟩	ASD region(s) - 5
☐	🟦	ASD region(s) - 6
☐	🟫	ASD region(s) - 7
☐	🟩	ASD region(s) - 8
☐	🟦	ASD region(s) - 9
☐	🟫	ASD region(s) - 10
☐	🟩	ASD region(s) - 11
	🟨	ASD region(s) - 1 and 2
	🟪	ASD region(s) - 1 and 3
	🟥	ASD region(s) - 1 and 4
	🟫	ASD region(s) - 1 and 5
	🟥	ASD region(s) - 1 and 6
	🟥	ASD region(s) - 1 and 7
	🟫	ASD region(s) - 1 and 8
	🟥	ASD region(s) - 1 and 9
	🟥	ASD region(s) - 1 and 10

16.3.3.3 Specifying Error Types

You can specify various types of errors for injection.

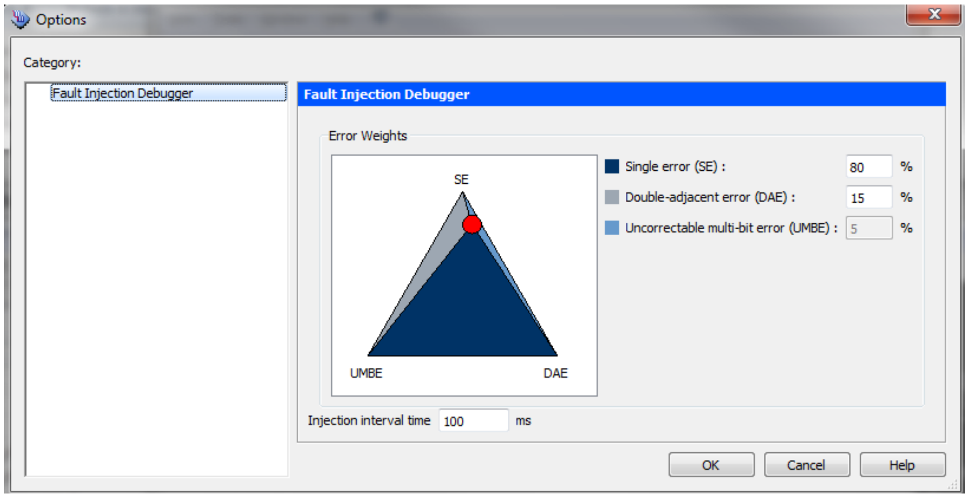
- Single errors (SE)
- Double-adjacent errors (DAE)
- Uncorrectable multi-bit errors (EMBE)

Intel FPGA devices can self-correct single and double-adjacent errors if the scrubbing feature is enabled. Intel FPGA devices cannot correct multi-bit errors. Refer to the chapter on mitigating SEUs for more information about debugging these errors.

You can specify the mixture of faults to inject and the injection time interval. To specify the injection time interval:

1. In the Fault Injection Debugger, choose **Tools ➤ Options**.
2. Drag the red controller to the mix of errors. Alternatively, you can specify the mix numerically.
3. Specify the **Injection interval time**.
4. Click **OK**.

Figure 224. Specifying the Mixture of SEU Fault Types



■ Single error (SE) :	80	%
■ Double-adjacent error (DAE) :	15	%
■ Uncorrectable multi-bit error (UMBE) :	5	%

Injection interval time 100 ms

Related Links

Mitigating Single Event Upset

16.3.3.4 Injecting Errors

You can inject errors in several modes:

- Inject one error on command
- Inject multiple errors on command
- Inject errors until commanded to stop

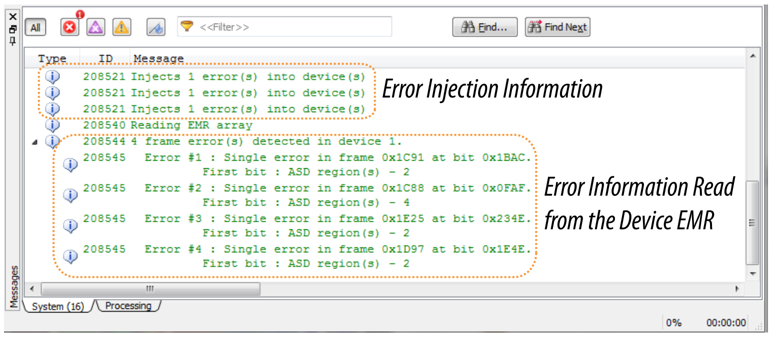
To inject these faults:

1. Turn on the **Inject Fault** option.
2. Choose whether you want to run error injection for a number of iterations or until stopped:
   - If you choose to run until stopped, the Fault Injection Debugger injects errors at the interval specified in the **Tools ➤ Options** dialog box.
   - If you want to run error injection for a specific number of iterations, enter the number.
3. Click **Start**.

   Note: The Fault Injection Debugger runs for the specified number of iterations or until stopped.

The Intel Quartus Prime Messages window shows messages about the errors that are injected. For additional information on the injected faults, click **Read EMR**. The Fault Injection Debugger reads the device's EMR and displays the contents in the Messages window.

Figure 225. Intel Quartus Prime Error Injection and EMR Content Messages


All
<<Filter>>
Find...
Find Next
Error Injection Information
Error Information Read from the Device EMR
Messages
System (16)
Processing
0%
00:00:00

```
Type    ID     Message
        208521 Injects 1 error(s) into device(s)
        208521 Injects 1 error(s) into device(s)
        208521 Injects 1 error(s) into device(s)
        208540 Reading EMR array
        208544 4 frame error(s) detected in device 1.
        208545   Error #1 : Single error in frame 0x1C91 at bit 0x1BAC.
                        First bit : ASD region(s) - 2
        208545   Error #2 : Single error in frame 0x1C88 at bit 0x0FAF.
                        First bit : ASD region(s) - 4
        208545   Error #3 : Single error in frame 0x1E25 at bit 0x234E.
                        First bit : ASD region(s) - 2
        208545   Error #4 : Single error in frame 0x1D97 at bit 0x1E4E.
                        First bit : ASD region(s) - 2
```

16.3.3.5 Recording Errors

You can record the location of any injected fault by noting the parameters reported in the Intel Quartus Prime Messages window.

If, for example, an injected fault results in behavior you would like to replay, you can target that location for injection. You perform targeted injection using the Fault Injection Debugger command line interface.

16.3.3.6 Clearing Injected Errors

To restore the normal function of the FPGA, click **Scrub**. When you scrub an error, the device's EDCRC functions are used to correct the errors. The scrub mechanism is similar to that used during device operation.

16.3.4 Command-Line Interface

You can run the Fault Injection Debugger at the command line with the **quartus_fid** executable, which is useful if you want to perform fault injection from a script.

Table 124. Command line Arguments for Fault Injection

Short Argument	Long Argument	Description
`c`	`cable`	Specify programming hardware or cable. (Required)
`i`	`index`	Specify the active device to inject fault. (Required)
`n`	`number`	Specify the number of errors to inject. The default value is 1. (Optional)
`t`	`time`	Interval time between injections. (Optional)

Note: Use `quartus_fid --help` to view all available options.

The following code provides examples using the Fault Injection Debugger command-line interface.

```
############################################
#
# Find out which USB cables are available for this instance
# The result shows that one cable is available, named "USB-Blaster"
#
$ quartus_fid --list
    . . .
    Info: Command: quartus_fid --list
    1) USB-Blaster on sj-sng-z4 [USB-0]
    Info: Intel Quartus Prime 64-Bit Fault Injection Debugger was successful.
     0 errors, 0 warning
############################################
#
# Find which devices are available on USB-Blaster cable
# The result shows two devices: a Stratix V A7, and a MAX V CPLD.
#
$ quartus_fid --cable USB-Blaster -a
    Info: Command: quartus_fid --cable=USB-Blaster -a
    Info (208809): Using programming cable "USB-Blaster on sj-sng-z4 [USB-0]"
    1) USB-Blaster on sj-sng-z4 [USB-0]
      029030DD   5SGXEA7H(1|2|3)/5SGXEA7K1/..
      020A40DD   5M2210Z/EPM2210
    Info: Intel Quartus Prime 64-Bit Fault Injection Debugger was successful.
    0 errors, 0 warnings

############################################
#
# Program the Stratix V device
# The --index option specifies operations performed on a connected device.
#   "=svgx.sof" associates a .sof file with the device
#   "#p" means program the device
#
$ quartus_fid --cable USB-Blaster --index "@1=svgx.sof#p"
 . . .
    Info (209016): Configuring device index 1
    Info (209017): Device 1 contains JTAG ID code 0x029030DD
    Info (209007): Configuration succeeded -- 1 device(s) configured
    Info (209011): Successfully performed operation(s)
    Info (208551): Program signature into device 1.
    Info: Intel Quartus Prime 64-Bit Fault Injection Debugger was successful.
    0 errors, 0 warnings

############################################
#
# Inject a fault into the device.
# The #i operator indicates to inject faults
# -n 3 indicates to inject 3 faults
#
$ quartus_fid --cable USB-Blaster --index "@1=svgx.sof#i" -n 3
    Info: Command: quartus_fid --cable=USB-Blaster --index=@1=svgx.sof#i -n 3
    Info (208809): Using programming cable "USB-Blaster on sj-sng-z4 [USB-0]"
    Info (208521): Injects 3 error(s) into device(s)
    Info: Intel Quartus Prime 64-Bit Fault Injection Debugger was successful.
    0 errors, 0 warnings


############################################
#
# Interactive Mode.
# Using the #i operation with -n 0 puts the debugger into interactive mode.
# Note that 3 faults were injected in the previous session;
# "E" reads the faults currently in the EMR Unloader IP core.
#
$ quartus_fid --cable USB-Blaster --index "@1=svgx.sof#i" -n 0
    Info: Command: quartus_fid --cable=USB-Blaster --index=@1=svgx.sof#i -n 0
    Info (208809): Using programming cable "USB-Blaster on sj-sng-z4 [USB-0]"
    Enter :
          'F' to inject fault
          'E' to read EMR
          'S' to scrub error(s)
          'Q' to quit
    E
    Info (208540): Reading EMR array
    Info (208544): 3 frame error(s) detected in device 1.
        Info (208545):   Error #1 : Single error in frame 0x1028 at bit
0x21EA.
        Info (10914):   Error #2 : Uncorrectable multi-bit error in frame
0x1116.
        Info (208545):   Error #3 : Single error in frame 0x1848 at bit
```

```
0x128C.
    Enter :
          'F' to inject fault
          'E' to read EMR
          'S' to scrub error(s)
          'Q' to quit
    Q
    Info: Intel Quartus Prime 64-Bit Fault Injection Debugger was successful.
        0 errors, 0 warnings
        Info: Peak virtual memory: 1522 megabytes
        Info: Processing ended: Mon Nov  3 18:50:00 2014
        Info: Elapsed time: 00:00:29
        Info: Total CPU time (on all processors): 00:00:13
```

16.3.4.1 Targeted Fault Injection Feature

The Fault Injection Debugger injects faults into the FPGA randomly. However, the Targeted Fault Injection feature allows you to inject faults into targeted locations in the CRAM. This operation may be useful, for example, if you noted an SEU event and want to test the FPGA or system response to the same event after modifying a recovery strategy.

Note: The Targeted Fault Injection feature is available only from the command line interface.

You can specify that errors are injected from the command line or in prompt mode.

Related Links

AN 539: Test Methodology or Error Detection and Recovery using CRC in Intel FPGA Devices

16.3.4.1.1 Specifying an Error List From the Command Line

The Targeted Fault Injection feature allows you to specify an error list from the command line, as shown in the following example:

```
c:\Users\sng> quartus_fid -c 1 - i "@1= svgx.sof#i " -n 2 -user="@1=
0x2274 0x05EF 0x2264 0x0500"
```
Where:

c 1 indicates that the fpga is controlled by the first cable on your computer.

`i "@1= svgx.sof#i "` indicates that the first device in the chain is loaded with the object file **svgx.sof** and will be injected with faults.

n 2 indicates that two faults will be injected.

`user=”@1= 0x2274 0x05EF 0x2264 0x0500”` is a user-specified list of faults to be injected. In this example, device 1 has two faults: at frame 0x2274, bit 0x05EF and at frame 0x2264, bit 0x0500.

16.3.4.1.2 Specifying an Error List From Prompt Mode

You can operate the Targeted Fault Injection feature interactively by specifying the number of faults to be 0 (-n 0). The Fault Injection Debugger presents prompt mode commands and their descriptions.

Prompt Mode Command	Description
F	Inject a fault
E	Read the EMR
S	Scrub errors
Q	Quit

In prompt mode, you can issue the F command alone to inject a single fault in a random location in the device. In the following examples using the F command in prompt mode, three errors are injected.

```
F #3 0x12 0x34 0x56 0x78 * 0x9A 0xBC +
```

- Error 1 – Single bit error at frame 0x12, bit 0x34
- Error 2 – Uncorrectable error at frame 0x56, bit 0x78 (an * indicates a multi-bit error)
- Error 3 – Double-adjacent error at frame 0x9A, bit 0xBC (a + indicates a double bit error)

```
F 0x12 0x34 0x56 0x78 *
```

One (default) error is injected:

Error 1 – Single bit error at frame 0x12, bit 0x34. Locations after the first frame/bit location are ignored.

```
F #3 0x12 0x34 0x56 0x78 * 0x9A 0xBC + 0xDE 0x00
```

Three errors are injected:

- Error 1 – Single bit error at frame 0x12, bit 0x34
- Error 2 – Uncorrectable error at frame 0x56, bit 0x78
- Error 3 – Double-adjacent error at frame 0x9A, bit 0xBC
- Locations after the first 3 frame/bit pairs are ignored

16.3.4.1.3 Determining CRAM Bit Locations

When the Fault Injection Debugger detects a CRAM EDCRC error, the Error Message Register (EMR) contains the syndrome, frame number, bit location, and error type (single, double, or multi-bit) of the detected CRAM error.

During system testing, save the EMR contents reported by the Fault Injection Debugger when you detect an EDCRC fault.

Note: With the recorded EMR contents, you can supply the frame and bit numbers to the Fault Injection Debugger to replay the errors noted during system testing, to further design, and characterize a system recovery response to that error.

Related Links

AN 539: Test Methodology or Error Detection and Recovery using CRC in Intel FPGA Devices

16.3.4.2 Advanced Command-Line Options: ASD Regions and Error Type Weighting

You can use the Fault Injection Debugger command-line interface to inject errors into ASD regions and weight the error types.

First, you specify the mix of error types (single bit, double adjacent, and multi-bit uncorrectable) using the `--weight` *`<single errors>`*`.`*`<double adjacent errors>`*`.`*`<multi-bit errors>`* option. For example, for a mix of 50% single errors, 30% double adjacent errors, and 20% multi-bit uncorrectable errors, use the option `--weight=50.30.20`. Then, to target an ASD region, use the `-smh` option to include the SMH file and indicate the ASD region to target. For example:

```
$ quartus_fid --cable=USB-BlasterII --index "@1=svgx.sof#pi" --
weight=100.0.0 --smh="@1=svgx.smh#2" --number=30
```

This example command:

- Programs the device and injects faults (`pi` string)
- Injects 100% single-bit faults (100.0.0)
- Injects only into ASD_REGION 2 (indicated by the #2)
- Injects 30 faults

16.4 Document Revision History

Table 125. Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
2015.05.04	15.0.0	• Provided more detail on how to use the Fault Injection Debugger throughout the document. • Added more command-line examples.
2014.06.30	14.0.0	• Removed "Modifying the Quartus INI File" section. • Added "Targeted Fault Injection Feature" section. • Updated "Hardware and Software Requirements" section.
December 2012	2012.12.01	Preliminary release.

Related Links

Documentation Archive
For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

17 In-System Debugging Using External Logic Analyzers

17.1 About the Intel Quartus Prime Logic Analyzer Interface

The Intel Quartus Prime Logic Analyzer Interface (LAI) allows you to use an external logic analyzer and a minimal number of Intel-supported device I/O pins to examine the behavior of internal signals while your design is running at full speed on your Intel-supported device.

The LAI connects a large set of internal device signals to a small number of output pins. You can connect these output pins to an external logic analyzer for debugging purposes. In the Intel Quartus Prime LAI, the internal signals are grouped together, distributed to a user-configurable multiplexer, and then output to available I/O pins on your Intel-supported device. Instead of having a one-to-one relationship between internal signals and output pins, the Intel Quartus Prime LAI enables you to map many internal signals to a smaller number of output pins. The exact number of internal signals that you can map to an output pin varies based on the multiplexer settings in the Intel Quartus Prime LAI.

Note: The term “logic analyzer” when used in this document includes both logic analyzers and oscilloscopes equipped with digital channels, commonly referred to as mixed signal analyzers or MSOs.

The LAI does not support Hard Processor System (HPS) I/Os.

Related Links

Device Support Center

17.2 Choosing a Logic Analyzer

The Intel Quartus Prime software offers the following two general purpose on-chip debugging tools for debugging a large set of RTL signals from your design:

- The Signal Tap Logic Analyzer
- An external logic analyzer, which connects to internal signals in your Intel-supported device by using the Intel Quartus Prime LAI

Table 126. Comparing the Signal Tap Logic Analyzer with the Logic Analyzer Interface

Feature	Description	Recommended Logic Analyzer
Sample Depth	You have access to a wider sample depth with an external logic analyzer. In the Signal Tap Logic Analyzer, the maximum sample depth is set to	LAI

continued...

Feature	Description	Recommended Logic Analyzer
	128 Kb, which is a device constraint. However, with an external logic analyzer, there are no device constraints, providing you a wider sample depth.	
Debugging Timing Issues	Using an external logic analyzer provides you with access to a “timing” mode, which enables you to debug combined streams of data.	LAI
Performance	You frequently have limited routing resources available to place and route when you use the Signal Tap Logic Analyzer with your design. An external logic analyzer adds minimal logic, which removes resource limits on place-and-route.	LAI
Triggering Capability	The Signal Tap Logic Analyzer offers triggering capabilities that are comparable to external logic analyzers.	LAI or Signal Tap
Use of Output Pins	Using the Signal Tap Logic Analyzer, no additional output pins are required. Using an external logic analyzer requires the use of additional output pins.	Signal Tap
Acquisition Speed	With the Signal Tap Logic Analyzer, you can acquire data at speeds of over 200 MHz. You can achieve the same acquisition speeds with an external logic analyzer; however, you must consider signal integrity issues.	Signal Tap

Related Links

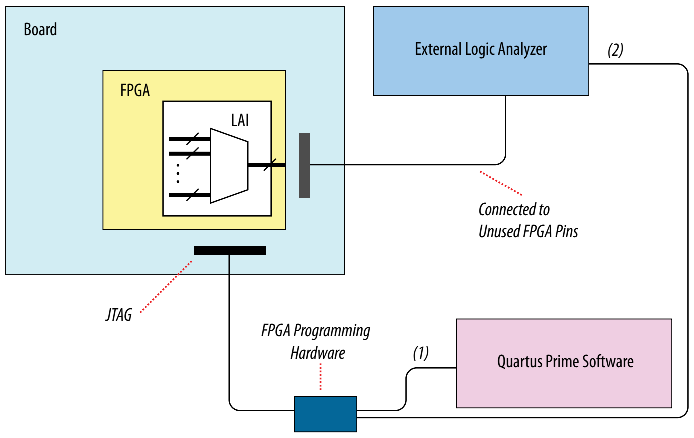
System Debugging Tools Overview on page 183

17.2.1 Required Components

To perform analysis using the LAI you need the following components:

- Intel Quartus Prime software version 15.1 or later
- The device under test
- An external logic analyzer
- An Intel FPGA communications cable
- A cable to connect the Intel-supported device to the external logic analyzer

Figure 226. LAI and Hardware Setup

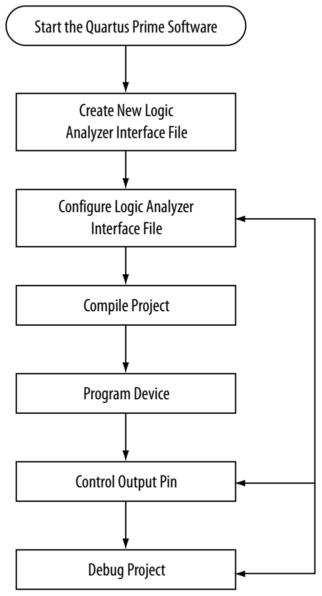


Notes to figure:

1. Configuration and control of the LAI using a computer loaded with the Intel Quartus Prime software via the JTAG port.
2. Configuration and control of the LAI using a third-party vendor logic analyzer via the JTAG port. Support varies by vendor.

17.3 Flow for Using the LAI

Figure 227. LAI Workflow


Start the Quartus Prime Software
Create New Logic Analyzer Interface File
Configure Logic Analyzer Interface File
Compile Project
Program Device
Control Output Pin
Debug Project

Notes to figure:

1. Configuration and control of the LAI using a computer loaded with the Intel Quartus Prime software via the JTAG port.
2. Configuration and control of the LAI using a third-party vendor logic analyzer via the JTAG port. Support varies by vendor.

17.4 Working with LAI Files

The `.lai` file stores the configuration of an LAI instance. The `.lai` file opens in the LAI editor. The editor allows you to group multiple internal signals to a set of external pins.

17.4.1 Configuring the File Core Parameters

After you create the `.lai` file, configure the `.lai` file core parameters by clicking on the **Setup View** list, and then selecting **Core Parameters**. The table below lists the `.lai` file core parameters.

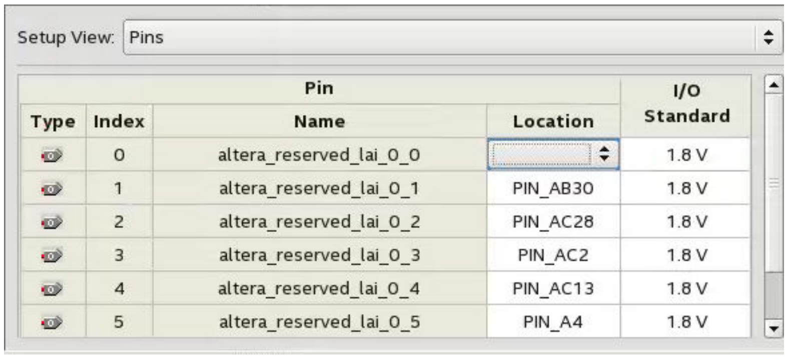
Table 127. LAI File Core Parameters

Parameter	Description
Pin Count	The **Pin Count** parameter signifies the number of pins you want dedicated to your LAI. The pins must be connected to a debug header on your board. Within the Intel-supported device, each pin is mapped to a user-configurable number of internal signals. The **Pin Count** parameter can range from 1 to 255 pins.
Bank Count	The **Bank Count** parameter signifies the number of internal signals that you want to map to each pin. For example, a **Bank Count** of 8 implies that you will connect eight internal signals to each pin. The **Bank Count** parameter can range from 1 to 255 banks.
Output/Capture Mode	The **Output/Capture Mode** parameter signifies the type of acquisition you perform. There are two options that you can select: **Combinational/Timing**—This acquisition uses your external logic analyzer's internal clock to determine when to sample data. Because **Combinational/Timing** acquisition samples data asynchronously to your Intel-supported device, you must determine the sample frequency you should use to debug and verify your system. This mode is effective if you want to measure timing information, such as channel-to-channel skew. For more information about the sampling frequency and the speeds at which it can run, refer to the data sheet for your external logic analyzer. **Registered/State**—This acquisition uses a signal from your system under test to determine when to sample. Because **Registered/State** acquisition samples data synchronously with your Intel-supported device, it provides you with a functional view of your Intel-supported device while it is running. This mode is effective when you verify the functionality of your design.
Clock	The **Clock** parameter is available only when **Output/Capture Mode** is set to **Registered State**. You must specify the sample clock in the **Core Parameters** view. The sample clock can be any signal in your design. However, for best results, Intel FPGA recommends that you use a clock with an operating frequency fast enough to sample the data you would like to acquire.
Power-Up State	The **Power-Up State** parameter specifies the power-up state of the pins you have designated for use with the LAI. You have the option of selecting tri-stated for all pins, or selecting a particular bank that you have enabled.

17.4.2 Mapping the LAI File Pins to Available I/O Pins

To configure the `.lai` file I/O pin parameters, select **Pins** in the **Setup View** list. To assign pin locations for the LAI, double-click the **Location** column next to the reserved pins in the **Name** column, and select a pin from the list. Once a pin is selected, you can right-click and locate in the Pin Planner.

Figure 228. Mapping LAI file Pins



Setup View: Pins

Pin				I/O Standard
Type	Index	Name	Location	
	0	altera_reserved_lai_0_0		1.8 V
	1	altera_reserved_lai_0_1	PIN_AB30	1.8 V
	2	altera_reserved_lai_0_2	PIN_AC28	1.8 V
	3	altera_reserved_lai_0_3	PIN_AC2	1.8 V
	4	altera_reserved_lai_0_4	PIN_AC13	1.8 V
	5	altera_reserved_lai_0_5	PIN_A4	1.8 V

Related Links

Managing Device I/O Pins
In *Intel Quartus Prime Standard Edition Volume 2*

17.4.3 Mapping Internal Signals to the LAI Banks

After you have specified the number of banks to use in the **Core Parameters** settings page, you must assign internal signals for each bank in the LAI. Click the **Setup View** arrow and select **Bank n** or **All Banks**.

To view all of your bank connections, click **Setup View** and select **All Banks**.

17.4.4 Using the Node Finder

Before making bank assignments, right click the Node list and select **Add Nodes** to open the **Node Finder**. Find the signals that you want to acquire, then drag and drop the signals from the **Node Finder** dialog box into the bank **Setup View**. When adding signals, use **Signal Tap: pre-synthesis** for non-incrementally routed instances and **Signal Tap: post-fitting** for incrementally routed instances.

As you continue to make assignments in the bank **Setup View**, the schematic of your LAI in the **Logical View** pane begins to reflect your changes. Continue making assignments for each bank in the **Setup View** until you have added all of the internal signals for which you wish to acquire data.

Related Links

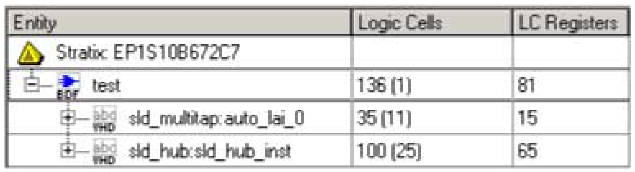
Node Finder Command
In Intel Quartus Prime Help

17.4.5 Compiling Your Intel Quartus Prime Project

After you save your `.lai` file, a dialog box prompts you to enable the Logic Analyzer Interface instance for the active project. Alternatively, you can define the `.lai` file your project uses in the **Global Project Settings** dialog box. After specifying the name of your `.lai` file, compile your project.

To verify the Logic Analyzer Interface is properly compiled with your project, expand the entity hierarchy in the Project Navigator. If the LAI is compiled with your design, the `sld_hub` and `sld_multitap` entities are shown in the Project Navigator.

Figure 229. Project Navigator



Entity	Logic Cells	LC Registers
Stratix: EP1S10B672C7		
test	136 (1)	81
sld_multitap:auto_lai_0	35 (11)	15
sld_hub:sld_hub_inst	100 (25)	65

17.4.6 Programming Your Intel-Supported Device Using the LAI

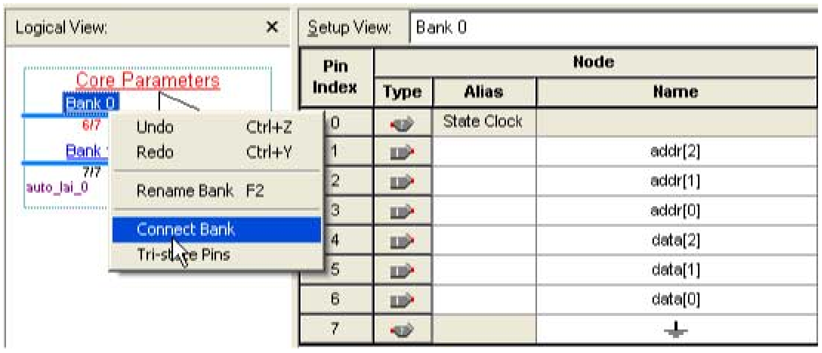
After compilation completes, you must configure your Intel-supported device before using the LAI.

You can use the LAI with multiple devices in your JTAG chain. Your JTAG chain can also consist of devices that do not support the LAI or non-Intel, JTAG-compliant devices. To use the LAI in more than one Intel-supported device, create an `.lai` file and configure an `.lai` file for each Intel-supported device that you want to analyze.

17.5 Controlling the Active Bank During Runtime

When you have programmed your Intel-supported device, you can control which bank you map to the reserved `.lai` file output pins. To control which bank you map, in the schematic in the Logical View, right-click the bank and click **Connect Bank**.

Figure 230. Configuring Banks



Setup View: Bank 0

Pin Index	Node		
	Type	Alias	Name
0		State Clock	
1			addr[2]
2			addr[1]
3			addr[0]
4			data[2]
5			data[1]
6			data[0]
7			

17.5.1 Acquiring Data on Your Logic Analyzer

To acquire data on your logic analyzer, you must establish a connection between your device and the external logic analyzer. For more information about this process and for guidelines about how to establish connections between debugging headers and logic analyzers, refer to the documentation for your logic analyzer.

17.6 Using the LAI with Incremental Compilation

The Incremental Compilation feature in the Intel Quartus Prime software allows you to preserve the synthesis and fitting results of your design. This is an effective feature for reducing compilation times if you only modify a portion of a design or you wish to preserve the optimization results from a previous compilation.

The Incremental Compilation feature is well suited for use with LAI since LAI comprises a small portion of most designs. Because LAI consists of only a small portion of your design, incremental compilation helps to minimize your compilation time. Incremental compilation works best when you are only changing a small portion of your design. Incremental compilation yields an accurate representation of your design behavior when changing the `.lai` file through multiple compilations.

17.7 Document Revision History

Table 128. Document Revision History

Date	Version	Changes
2017.05.08	17.0.0	• Updated figure: LAI Instance in Compilation Report.
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
June 2014	14.0.0	• Dita conversion • Added limitation about HPS I/O support
June 2012	12.0.0	Removed survey link
November 2011	10.1.1	Changed to new document template
December 2010	10.1.0	• Minor editorial updates • Changed to new document template
August 2010	10.0.1	Corrected links
July 2010	10.0.0	• Created links to the Intel Quartus Prime Help • Editorial updates • Removed Referenced Documents section
November 2009	9.1.0	• Removed references to APEX devices • Editorial updates
March 2009	9.0.0	• Minor editorial updates • Removed Figures 15–4, 15–5, and 15–11 from 8.1 version
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content
May 2008	8.0.0	• Updated device support list on page 15–3 • Added links to referenced documents throughout the chapter • Added "Referenced Documents" • Added reference to *Section V. In-System Debugging* • Minor editorial updates

Related Links

Documentation Archive
For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

18 In-System Modification of Memory and Constants

18.1 About the In-System Memory Content Editor

The Intel Quartus Prime In-System Memory Content Editor allows to view and update memories and constants with the JTAG port connection at runtime.

The In-System Memory Content Editor provides access to dense and complex FPGA designs through the JTAG interface. You can then identify, test, and resolve issues with your design by testing changes to memory contents in the FPGA while your design is running.

When you use the In-System Memory Content Editor in conjunction with the Signal Tap Logic Analyzer, you can view and debug your design in the hardware lab.

The ability to read data from memories and constants allows you to identify the source of problems. The write capability allows you to bypass functional issues by writing expected data. For example, if a parity bit in your memory is incorrect, you can use the In-System Memory Content Editor to write the correct parity bit values into your RAM, allowing your system to continue functioning. To check the error handling functionality of your design, you can intentionally write incorrect parity bit values into your RAM.

Related Links

- System Debugging Tools Overview on page 183
- Design Debugging with the Signal Tap Logic Analyzer on page 327
- Megafunctions/LPM
  List of the types of memories and constants currently supported by the Intel Quartus Prime software

18.2 Design Flow Using the In-System Memory Content Editor

To use the In-System Memory Content Editor, perform the following steps:

1. Identify the memories and constants that you want to access.
2. Edit the memories and constants to be run-time modifiable.
3. Perform a full compilation.
4. Program your device.
5. Launch the In-System Memory Content Editor.

18.3 Creating In-System Modifiable Memories and Constants

When you specify that a memory or constant is run-time modifiable, the Intel Quartus Prime software changes the default implementation. A single-port RAM is converted to a dual-port RAM, and a constant is implemented in registers instead of look-up tables (LUTs). These changes enable run-time modification without changing the functionality of your design.

If you instantiate a memory or constant IP core directly with ports and parameters in VHDL or Verilog HDL, add or modify the `lpm_hint` parameter as follows:

In VHDL code, add the following:

```
lpm_hint => "ENABLE_RUNTIME_MOD = YES,
   INSTANCE_NAME = <instantiation name>";
```

In Verilog HDL code, add the following:

```
defparam <megafunction instance name>.lpm_hint =
   "ENABLE_RUNTIME_MOD = YES,
   INSTANCE_NAME = <instantiation name>";
```

18.4 Running the In-System Memory Content Editor

The In-System Memory Content Editor has three separate panes: the **Instance Manager**, the **JTAG Chain Configuration**, and the **Hex Editor**.

The **Instance Manager** pane displays all available run-time modifiable memories and constants in your FPGA device. The **JTAG Chain Configuration** pane allows you to program your FPGA and select the Intel FPGA device in the chain to update.

Using the In-System Memory Content Editor does not require that you open a project. The In-System Memory Content Editor retrieves all instances of run-time configurable memories and constants by scanning the JTAG chain and sending a query to the specific device selected in the **JTAG Chain Configuration** pane.

If you have more than one device with in-system configurable memories or constants in a JTAG chain, you can launch multiple In-System Memory Content Editors within the Intel Quartus Prime software to access the memories and constants in each of the devices. Each In-System Memory Content Editor can access the in-system memories and constants in a single device.

18.4.1 Instance Manager

You can read and write to in-system memory with the **Instance Manager** pane. When you scan the JTAG chain to update the **Instance Manager** pane, you can view a list of all run-time modifiable memories and constants in the design. The **Instance Manager** pane displays the Index, Instance, Status, Width, Depth, Type, and Mode of each element in the list.

Note: In addition to the buttons available in the **Instance Manager** pane, you can read and write data by selecting commands from the **Processing** menu, or the right-click menu in the **Instance Manager** pane or **Hex Editor** pane.

The status of each instance is also displayed beside each entry in the **Instance Manager** pane. The status indicates if the instance is **Not running**, **Offloading data**, or **Updating data**. The health monitor provides information about the status of the editor.

The Intel Quartus Prime software assigns a different index number to each in-system memory and constant to distinguish between multiple instances of the same memory or constant function. View the **In-System Memory Content Editor Settings** section of the Compilation Report to match an index number with the corresponding instance ID.

Related Links

Instance Manager Pane
In Intel Quartus Prime Help

18.4.2 Editing Data Displayed in the Hex Editor Pane

You can edit data read from your in-system memories and constants displayed in the **Hex Editor** pane by typing values directly into the editor or by importing memory files.

18.4.3 Importing and Exporting Memory Files

The In-System Memory Content Editor allows you to import and export data values for memories that have the In-System Updating feature enabled. Importing from a data file enables you to quickly load an entire memory image. Exporting to a data file enables you to save the contents of the memory for future use.

18.4.4 Scripting Support

The In-System Memory Content Editor supports reading and writing of memory contents via a Tcl script or Tcl commands entered at a command prompt. For detailed information about scripting command options, refer to the Intel Quartus Prime command-line and Tcl API Help browser.

To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp
```

The commonly used commands for the In-System Memory Content Editor are as follows:

- Reading from memory:

```
read_content_from_memory
[-content_in_hex]
-instance_index <instance index>
-start_address <starting address>
-word_count <word count>
```

- Writing to memory:

  `write_content_to_memory`
- Saving memory contents to a file:

  `save_content_from_memory_to_file`
- Updating memory contents from a file:

  `update_content_to_memory_from_file`

Related Links

- Tcl Scripting
  In *Intel Quartus Prime Standard Edition Handbook Volume 2*
- Command Line Scripting
  In *Intel Quartus Prime Standard Edition Handbook Volume 2*
- API Functions for Tcl
  In Intel Quartus Prime Help

18.4.5 Programming the Device with the In-System Memory Content Editor

If you make changes to your design, you can program the device from within the In-System Memory Content Editor.

18.4.6 Example: Using the In-System Memory Content Editor with the Signal Tap Logic Analyzer

The following scenario describes how you can use the In-System Updating of Memory and Constants feature with the Signal Tap Logic Analyzer to efficiently debug your design. You can use the In-System Memory Content Editor and the Signal Tap Logic Analyzer simultaneously with the JTAG interface.

Scenario: After completing your FPGA design, you find that the characteristics of your FIR filter design are not as expected.

1. To locate the source of the problem, change all your FIR filter coefficients to be in-system modifiable and instantiate the Signal Tap Logic Analyzer.
2. Using the Signal Tap Logic Analyzer to tap and trigger on internal design nodes, you find the FIR filter to be functioning outside of the expected cutoff frequency.
3. Using the **In-System Memory Content Editor**, you check the correctness of the FIR filter coefficients. Upon reading each coefficient, you discover that one of the coefficients is incorrect.
4. Because your coefficients are in-system modifiable, you update the coefficients with the correct data with the **In-System Memory Content Editor**.

In this scenario, you can quickly locate the source of the problem using both the In-System Memory Content Editor and the Signal Tap Logic Analyzer. You can also verify the functionality of your device by changing the coefficient values before modifying the design source files.

You can also modify the coefficients with the In-System Memory Content Editor to vary the characteristics of the FIR filter, for example, filter attenuation, transition bandwidth, cut-off frequency, and windowing function.

18.5 Document Revision History

Table 129. Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
June 2014	14.0.0	• Dita conversion. • Removed references to megafunction and replaced with IP core.
June 2012	12.0.0	Removed survey link.
November 2011	10.0.3	Template update.
December 2010	10.0.2	Changed to new document template. No change to content.
August 2010	10.0.1	Corrected links
July 2010	10.0.0	• Inserted links to Intel Quartus Prime Help • Removed Reference Documents section
November 2009	9.1.0	• Delete references to APEX devices • Style changes
March 2009	9.0.0	No change to content
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	8.0.0	• Added reference to Section V. In-System Debugging in volume 3 of the Intel Quartus Prime Handbook on page 16-1 • Removed references to the Mercury device, as it is now considered to be a "Mature" device • Added links to referenced documents throughout document • Minor editorial updates

Related Links

Documentation Archive
For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

19 Design Debugging Using In-System Sources and Probes

The Signal Tap Logic Analyzer and Signal Probe allow you to read or “tap” internal logic signals during run time as a way to debug your logic design.

Traditional debugging techniques often involve using an external pattern generator to exercise the logic and a logic analyzer to study the output waveforms during run time.

You can make the debugging cycle more efficient when you can drive any internal signal manually within your design, which allows you to perform the following actions:

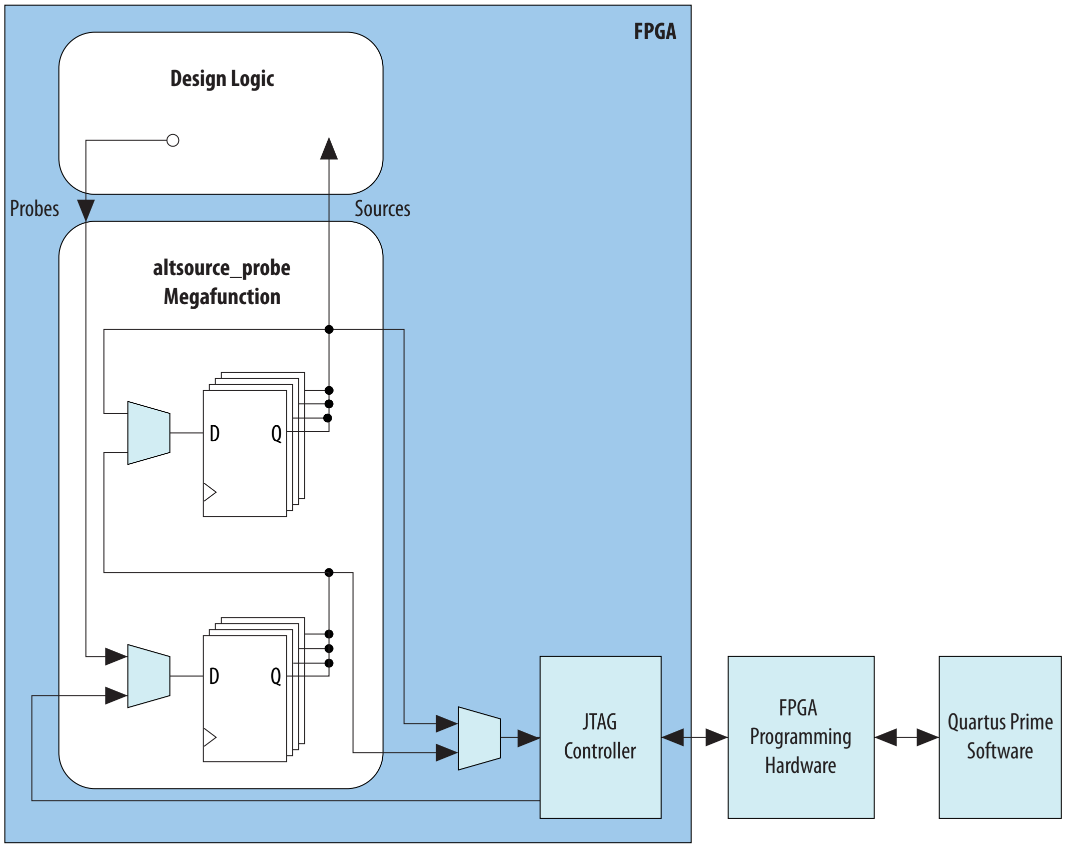
- Force the occurrence of trigger conditions set up in the Signal Tap Logic Analyzer
- Create simple test vectors to exercise your design without using external test equipment
- Dynamically control run time control signals with the JTAG chain

The In-System Sources and Probes Editor in the Intel Quartus Prime software extends the portfolio of verification tools, and allows you to easily control any internal signal and provides you with a completely dynamic debugging environment. Coupled with either the Signal Tap Logic Analyzer or Signal Probe, the In-System Sources and Probes Editor gives you a powerful debugging environment in which to generate stimuli and solicit responses from your logic design.

The Virtual JTAG IP core and the In-System Memory Content Editor also give you the capability to drive virtual inputs into your design. The Intel Quartus Prime software offers a variety of on-chip debugging tools.

The In-System Sources and Probes Editor consists of the ALTSOURCE_PROBE IP core and an interface to control the ALTSOURCE_PROBE IP core instances during run time. Each ALTSOURCE_PROBE IP core instance provides you with source output ports and probe input ports, where source ports drive selected signals and probe ports sample selected signals. When you compile your design, the ALTSOURCE_PROBE IP core sets up a register chain to either drive or sample the selected nodes in your logic design. During run time, the In-System Sources and Probes Editor uses a JTAG connection to shift data to and from the ALTSOURCE_PROBE IP core instances. The figure shows a block diagram of the components that make up the In-System Sources and Probes Editor.

Figure 231. In-System Sources and Probes Editor Block Diagram



The ALTSOURCE_PROBE IP core hides the detailed transactions between the JTAG controller and the registers instrumented in your design to give you a basic building block for stimulating and probing your design. Additionally, the In-System Sources and Probes Editor provides single-cycle samples and single-cycle writes to selected logic nodes. You can use this feature to input simple virtual stimuli and to capture the current value on instrumented nodes. Because the In-System Sources and Probes Editor gives you access to logic nodes in your design, you can toggle the inputs of low-level components during the debugging process. If used in conjunction with the Signal Tap Logic Analyzer, you can force trigger conditions to help isolate your problem and shorten your debugging process.

The In-System Sources and Probes Editor allows you to easily implement control signals in your design as virtual stimuli. This feature can be especially helpful for prototyping your design, such as in the following operations:

- Creating virtual push buttons
- Creating a virtual front panel to interface with your design
- Emulating external sensor data
- Monitoring and changing run time constants on the fly

The In-System Sources and Probes Editor supports Tcl commands that interface with all your ALTSOURCE_PROBE IP core instances to increase the level of automation.

Related Links

- System Debugging Tools
  For an overview and comparison of all the tools available in the Intel Quartus Prime software on-chip debugging tool suite
- System Debugging Tools
  For an overview and comparison of all the tools available in the Intel Quartus Prime software on-chip debugging tool suite

19.1 Hardware and Software Requirements

The following components are required to use the In-System Sources and Probes Editor:

- Intel Quartus Prime software

or

- Intel Quartus Prime Lite Edition
- Download Cable (USB-Blaster™ download cable or ByteBlaster™ cable)
- Intel FPGA development kit or user design board with a JTAG connection to device under test

The In-System Sources and Probes Editor supports the following device families:

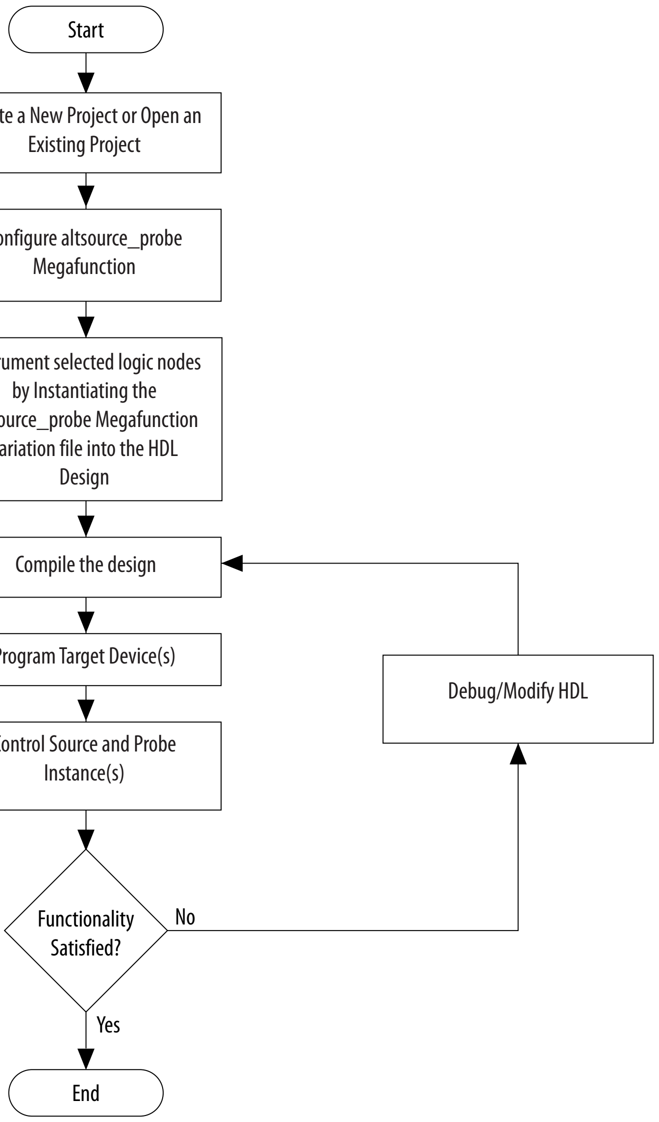
- Arria® series
- Stratix® series
- Cyclone® series
- MAX® series

19.2 Design Flow Using the In-System Sources and Probes Editor

The In-System Sources and Probes Editor supports an RTL flow. Signals that you want to view in the In-System Sources and Probes editor are connected to an instance of the In-System Sources and Probes IP core.

After you compile the design, you can control each instance via the **In-System Sources and Probes Editor** pane or via a Tcl interface.

Figure 232. FPGA Design Flow Using the In-System Sources and Probes Editor


Start
Create a New Project or Open an Existing Project
Configure altsource_probe Megafunction
Instrument selected logic nodes by Instantiating the altsource_probe Megafunction variation file into the HDL Design
Compile the design
Program Target Device(s)
Control Source and Probe Instance(s)
Functionality Satisfied?
No
Yes
Debug/Modify HDL
End

19.2.1 Instantiating the In-System Sources and Probes IP Core

You must instantiate the In-System Sources and Probes IP core before you can use the In-System Sources and Probes editor. Use the IP Catalog and parameter editor to instantiate a custom variation of the In-System Sources and Probes IP core.

To configure the In-System Sources and Probes IP core, perform the following steps::

1. On the Tools menu, click **Tools > IP Catalog**.
2. Locate and double-click the In-System Sources and Probes IP core. The parameter editor appears.
3. Specify a name for your custom IP variation.
4. Specify the desired parameters for your custom IP variation. You can specify up to up to 512 bits for each source. Your design may include up to 128 instances of this IP core.
5. Click **Generate** or **Finish** to generate IP core synthesis and simulation files matching your specifications. The parameter editor generates the necessary variation files and the instantiation template based on your specification. Use the generated template to instantiate the In-System Sources and Probes IP core in your design.

   Note: The In-System Sources and Probes Editor does not support simulation. You must remove the In-System Sources and Probes IP core before you create a simulation netlist.

19.2.2 In-System Sources and Probes IP Core Parameters

Use the template to instantiate the variation file in your design.

Table 130. In-System Sources and Probes IP Port Information

Port Name	Required?	Direction	Comments
`probe[]`	No	Input	The outputs from your design.
`source_clk`	No	Input	Source Data is written synchronously to this clock. This input is required if you turn on **Source Clock** in the **Advanced Options** box in the parameter editor.
`source_ena`	No	Input	Clock enable signal for `source_clk`. This input is required if specified in the **Advanced Options** box in the parameter editor.
`source[]`	No	Output	Used to drive inputs to user design.

You can include up to 128 instances of the in-system sources and probes IP core in your design, if your device has available resources. Each instance of the IP core uses a pair of registers per signal for the width of the widest port in the IP core. Additionally, there is some fixed overhead logic to accommodate communication between the IP core instances and the JTAG controller. You can also specify an additional pair of registers per source port for synchronization.

You can use the Intel Quartus Prime incremental compilation feature to reduce compilation time. Incremental compilation allows you to organize your design into logical partitions. During recompilation of a design, incremental compilation preserves the compilation results and performance of unchanged partitions and reduces design iteration time by compiling only modified design partitions.

19.3 Compiling the Design

When you compile your design that includes the In-System Sources and ProbesIP core, the In-System Sources and Probes and SLD Hub Controller IP core are added to your compilation hierarchy automatically. These IP cores provide communication between the JTAG controller and your instrumented logic.

You can modify the number of connections to your design by editing the In-System Sources and Probes IP core. To open the design instance you want to modify in the parameter editor, double-click the instance in the Project Navigator. You can then modify the connections in the HDL source file. You must recompile your design after you make changes.

You can use the Intel Quartus Prime incremental compilation feature to reduce compilation design into logical partitions. During recompilation of a design, incremental compilation preserves the compilation results and performance of unchanged partitions and reduces design iteration time by compiling only modified design partitions.

19.4 Running the In-System Sources and Probes Editor

The In-System Sources and Probes Editor gives you control over all ALTSOURCE_PROBE IP core instances within your design. The editor allows you to view all available run time controllable instances of the ALTSOURCE_PROBE IP core in your design, provides a push-button interface to drive all your source nodes, and provides a logging feature to store your probe and source data.

To run the In-System Sources and Probes Editor:

- On the **Tools** menu, click **In-System Sources and Probes Editor.**

19.4.1 In-System Sources and Probes Editor GUI

The In-System Sources and Probes Editor contains three panes:

- **JTAG Chain Configuration**—Allows you to specify programming hardware, device, and file settings that the In-System Sources and Probes Editor uses to program and acquire data from a device.
- **Instance Manager**—Displays information about the instances generated when you compile a design, and allows you to control data that the In-System Sources and Probes Editor acquires.
- **In-System Sources and Probes Editor**—Logs all data read from the selected instance and allows you to modify source data that is written to your device.

When you use the In-System Sources and Probes Editor, you do not need to open a Intel Quartus Prime software project. The In-System Sources and Probes Editor retrieves all instances of the ALTSOURCE_PROBE IP core by scanning the JTAG chain and sending a query to the device selected in the **JTAG Chain Configuration** pane. You can also use a previously saved configuration to run the In-System Sources and Probes Editor.

Each **In-System Sources and Probes Editor** pane can access the ALTSOURCE_PROBE IP core instances in a single device. If you have more than one device containing IP core instances in a JTAG chain, you can launch multiple **In-System Sources and Probes Editor** panes to access the IP core instances in each device.

19.4.2 Programming Your Device With JTAG Chain Configuration

After you compile your project, you must configure your FPGA before you use the In-System Sources and Probes Editor.

To configure a device to use with the In-System Sources and Probes Editor, perform the following steps:

1. Open the In-System Sources and Probes Editor.
2. In the **JTAG Chain Configuration** pane, point to **Hardware,** and then select the hardware communications device. You may be prompted to configure your hardware; in this case, click **Setup**.
3. From the **Device** list, select the FPGA device to which you want to download the design (the device may be automatically detected). You may need to click **Scan Chain** to detect your target device.
4. In the **JTAG Chain Configuration** pane, click to browse for the SRAM Object File (**.sof**) that includes the In-System Sources and Probes instance or instances. (The **.sof** may be automatically detected).
5. Click **Program Device** to program the target device.

19.4.3 Instance Manager

The **Instance Manager** pane provides a list of all ALTSOURCE_PROBE instances in the design and allows you to configure how data is acquired from or written to those instances.

The following buttons and sub-panes are provided in the **Instance Manager** pane:

- **Read Probe Data**—Samples the probe data in the selected instance and displays the probe data in the **In-System Sources and Probes Editor** pane.
- **Continuously Read Probe Data**—Continuously samples the probe data of the selected instance and displays the probe data in the **In-System Sources and Probes Editor** pane; you can modify the sample rate via the **Probe read interval** setting.
- **Stop Continuously Reading Probe Data**—Cancels continuous sampling of the probe of the selected instance.
- **Read Source Data**—Reads the data of the sources in the selected instances.
- **Probe Read Interval**—Displays the sample interval of all the In-System Sources and Probe instances in your design; you can modify the sample interval by clicking **Manual.**
- **Event Log**—Controls the event log in the **In-System Sources and Probes Editor** pane.
- **Write Source Data**—Allows you to manually or continuously write data to the system.

The status of each instance is also displayed beside each entry in the **Instance Manager** pane. The status indicates if the instance is **Not running Offloading data**, **Updating data**, or if an **Unexpected JTAG communication error** occurs. This status indicator provides information about the sources and probes instances in your design.

19.4.4 In-System Sources and Probes Editor Pane

The **In-System Sources and Probes Editor** pane allows you to view data from all sources and probes in your design.

The data is organized according to the index number of the instance. The editor provides an easy way to manage your signals, and allows you to rename signals or group them into buses. All data collected from in-system source and probe nodes is recorded in the event log and you can view the data as a timing diagram.

19.4.4.1 Reading Probe Data

You can read data by selecting the ALTSOURCE_PROBE instance in the **Instance Manager** pane and clicking **Read Probe Data**.

This action produces a single sample of the probe data and updates the data column of the selected index in the **In-System Sources and Probes Editor** pane. You can save the data to an event log by turning on the **Save data to event log** option in the **Instance Manager** pane.

If you want to sample data from your probe instance continuously, in the **Instance Manager** pane, click the instance you want to read, and then click **Continuously read probe data**. While reading, the status of the active instance shows **Unloading**. You can read continuously from multiple instances.

You can access read data with the shortcut menus in the **Instance Manager** pane.

To adjust the probe read interval, in the **Instance Manager** pane, turn on the **Manual** option in the **Probe read interval** sub-pane, and specify the sample rate in the text field next to the **Manual** option. The maximum sample rate depends on your computer setup. The actual sample rate is shown in the **Current interval** box. You can adjust the event log window buffer size in the **Maximum Size** box.

19.4.4.2 Writing Data

To modify the source data you want to write into the ALTSOURCE_PROBE instance, click the name field of the signal you want to change. For buses of signals, you can double-click the data field and type the value you want to drive out to the ALTSOURCE_PROBE instance. The In-System Sources and Probes Editor stores the modified source data values in a temporary buffer.

Modified values that are not written out to the ALTSOURCE_PROBE instances appear in red. To update the ALTSOURCE_PROBE instance, highlight the instance in the **Instance Manager** pane and click **Write source data**. The **Write source data** function is also available via the shortcut menus in the **Instance Manager** pane.

The In-System Sources and Probes Editor provides the option to continuously update each ALTSOURCE_PROBE instance. Continuous updating allows any modifications you make to the source data buffer to also write immediately to the ALTSOURCE_PROBE instances. To continuously update the ALTSOURCE_PROBE instances, change the **Write source data** field from **Manually** to **Continuously**.

19.4.4.3 Organizing Data

The **In-System Sources and Probes Editor** pane allows you to group signals into buses, and also allows you to modify the display options of the data buffer.

To create a group of signals, select the node names you want to group, right-click and select **Group**. You can modify the display format in the Bus Display Format and the Bus Bit order shortcut menus.

The **In-System Sources and Probes Editor** pane allows you to rename any signal. To rename a signal, double-click the name of the signal and type the new name.

The event log contains a record of the most recent samples. The buffer size is adjustable up to 128k samples. The time stamp for each sample is logged and is displayed above the event log of the active instance as you move your pointer over the data samples.

You can save the changes that you make and the recorded data to a Sources and Probes File (`.spf`). To save changes, on the File menu, click **Save**. The file contains all the modifications you made to the signal groups, as well as the current data event log.

19.5 Tcl interface for the In-System Sources and Probes Editor

To support automation, the In-System Sources and Probes Editor supports the procedures described in this chapter in the form of Tcl commands. The Tcl package for the In-System Sources and Probes Editor is included by default when you run **quartus_stp**.

The Tcl interface for the In-System Sources and Probes Editor provides a powerful platform to help you debug your design. The Tcl interface is especially helpful for debugging designs that require toggling multiple sets of control inputs. You can combine multiple commands with a Tcl script to define a custom command set.

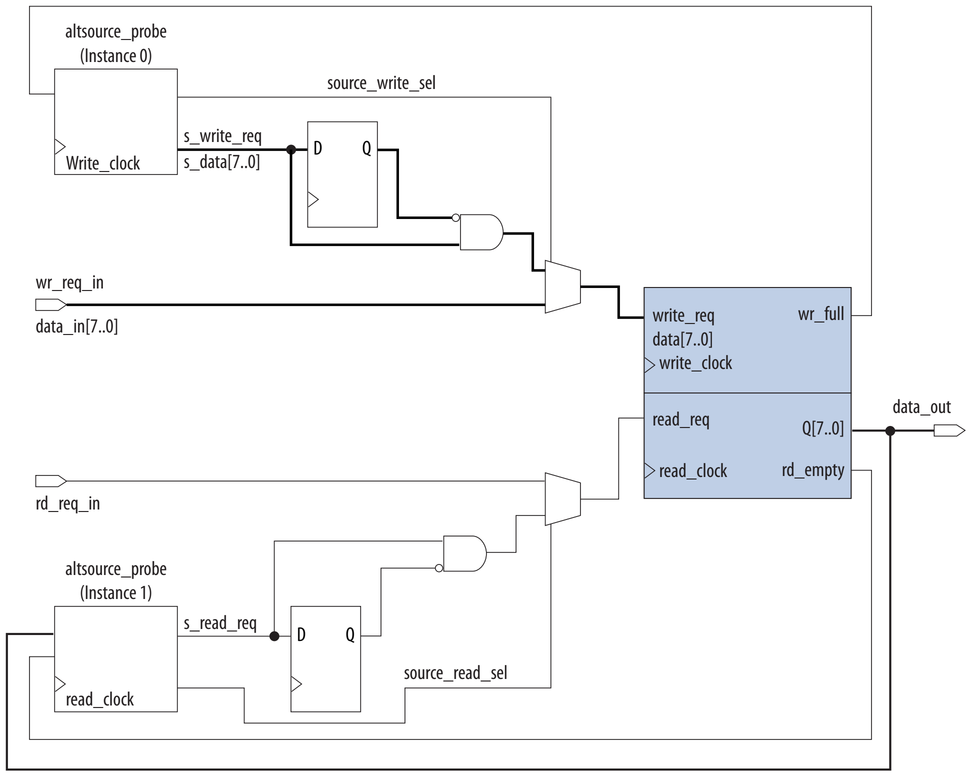
Table 131. In-System Sources and Probes Tcl Commands

Command	Argument	Description
`start_insystem_source_probe`	`-device_name` *<device name>* `-hardware_name` *<hardware name>*	Opens a handle to a device with the specified hardware. Call this command before starting any transactions.
`get_insystem_source_probe_instance_info`	`-device_name` *<device name>* -hardware_name *<hardware name>*	Returns a list of all ALTSOURCE_PROBE instances in your design. Each record returned is in the following format: {*<instance Index>*, *<source width>*, *<probe width>*, *<instance name>*}
`read_probe_data`	`-instance_index` *<instance_index>* `-value_in_hex` (optional)	Retrieves the current value of the probe. A string is returned that specifies the status of each probe, with the MSB as the left-most bit.
`read_source_data`	`-instance_index` *<instance_index>* `-value_in_hex` (optional)	Retrieves the current value of the sources. A string is returned that specifies the status of each source, with the MSB as the left-most bit.
`write_source_data`	`-instance_index` *<instance_index>* `-value` *<value>* `-value_in_hex` (optional)	Sets the value of the sources. A binary string is sent to the source ports, with the MSB as the left-most bit.
`end_insystem_source_probe`	None	Releases the JTAG chain. Issue this command when all transactions are finished.

The example shows an excerpt from a Tcl script with procedures that control the ALTSOURCE_PROBE instances of the design as shown in the figure below. The example design contains a DCFIFO with ALTSOURCE_PROBE instances to read from and write to the DCFIFO. A set of control muxes are added to the design to control the flow of data to the DCFIFO between the input pins and the ALTSOURCE_PROBE instances. A pulse generator is added to the read request and write request control lines to guarantee a single sample read or write. The ALTSOURCE_PROBE instances, when used with the script in the example below, provide visibility into the contents of the FIFO by performing single sample write and read operations and reporting the state of the full and empty status flags.

Use the Tcl script in debugging situations to either empty or preload the FIFO in your design. For example, you can use this feature to preload the FIFO to match a trigger condition you have set up within the Signal Tap Logic Analyzer.

Figure 233. DCFIFO Example Design Controlled by Tcl Script



```
## Setup USB hardware  - assumes only USB Blaster is installed and
## an FPGA is the only device in the JTAG chain
set usb [lindex [get_hardware_names] 0]
set device_name [lindex [get_device_names -hardware_name $usb] 0]
## write procedure :  argument value is integer
proc write {value} {
global device_name usb
variable full
start_insystem_source_probe -device_name $device_name -hardware_name $usb
#read full flag
set full [read_probe_data -instance_index 0]
if {$full == 1} {end_insystem_source_probe
return "Write Buffer Full"
}
##toggle select line, drive value onto port, toggle enable
##bits 7:0 of instance 0 is S_data[7:0]; bit 8 = S_write_req;
##bit 9 = Source_write_sel
##int2bits is custom procedure that returns a bitstring from an integer
    ## argument
write_source_data -instance_index 0 -value /[int2bits [expr 0x200 | $value]]
write_source_data -instance_index 0 -value [int2bits [expr 0x300 | $value]]
##clear transaction
write_source_data -instance_index 0 -value 0
end_insystem_source_probe
}
proc read {} {
global device_name usb
variable empty
start_insystem_source_probe -device_name $device_name -hardware_name $usb
##read empty flag : probe port[7:0] reads FIFO output; bit 8 reads empty_flag
set empty [read_probe_data -instance_index 1]
if {[regexp {1........} $empty]} { end_insystem_source_probe
return "FIFO empty" }
## toggle select line for read transaction
## Source_read_sel = bit 0; s_read_reg = bit 1
## pulse read enable on DC FIFO
write_source_data -instance_index 1 -value 0x1 -value_in_hex
write_source_data -instance_index 1 -value 0x3 -value_in_hex
set x [read_probe_data -instance_index 1 ]
end_insystem_source_probe
return $x
}
```

Related Links

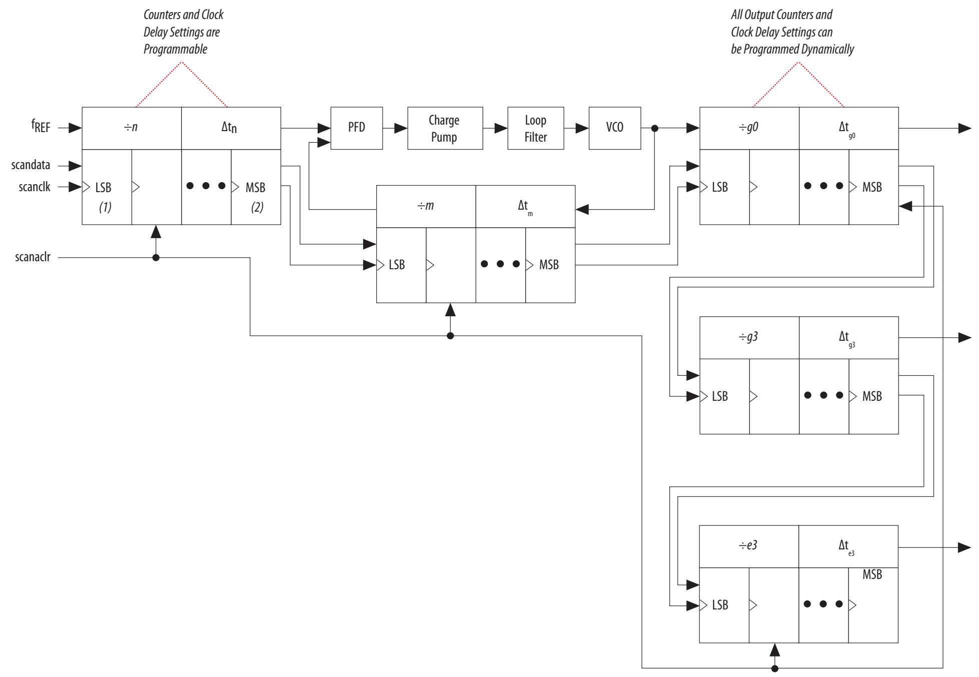
- Tcl Scripting
- Intel Quartus Prime Settings File Manual
- Command Line Scripting
- Tcl Scripting
- Intel Quartus Prime Settings File Manual
- Command Line Scripting

19.6 Design Example: Dynamic PLL Reconfiguration

The In-System Sources and Probes Editor can help you create a virtual front panel during the prototyping phase of your design. You can create relatively simple, high functioning designs of in a short amount of time. The following PLL reconfiguration example demonstrates how to use the In-System Sources and Probes Editor to provide a GUI to dynamically reconfigure a Stratix PLL.

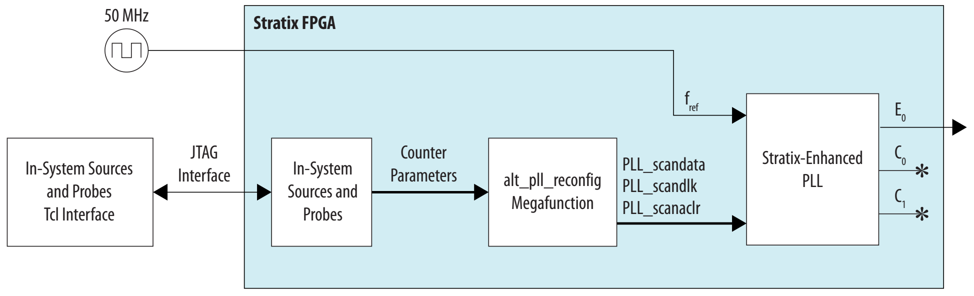
Stratix PLLs allow you to dynamically update PLL coefficients during run time. Each enhanced PLL within the Stratix device contains a register chain that allows you to modify the pre-scale counters (m and n values), output divide counters, and delay counters. In addition, the ALTPLL_RECONFIG IP core provides an easy interface to access the register chain counters. The ALTPLL_RECONFIG IP core provides a cache that contains all modifiable PLL parameters. After you update all the PLL parameters in the cache, the ALTPLL_RECONFIG IP core drives the PLL register chain to update the PLL with the updated parameters. The figure shows a Stratix-enhanced PLL with reconfigurable coefficients.

Figure 234. Stratix-Enhanced PLL with Reconfigurable Coefficients


Counters and Clock Delay Settings are Programmable
All Output Counters and Clock Delay Settings can be Programmed Dynamically
fREF
scandata
scanclk
scanaclr
÷n
Δtn
LSB (1)
MSB (2)
PFD
Charge Pump
Loop Filter
VCO
÷m
Δtm
LSB
MSB
÷g0
Δtg0
÷g3
Δtg3
÷e3
Δte3

The following design example uses an ALTSOURCE_PROBE instance to update the PLL parameters in the ALTPLL_RECONFIG IP core cache. The ALTPLL_RECONFIG IP core connects to an enhanced PLL in a Stratix FPGA to drive the register chain containing the PLL reconfigurable coefficients. This design example uses a Tcl/Tk script to generate a GUI where you can enter in new m and n values for the enhanced PLL. The Tcl script extracts the m and n values from the GUI, shifts the values out to the ALTSOURCE_PROBE instances to update the values in the ALTPLL_RECONFIG IP core cache, and asserts the reconfiguration signal on the ALTPLL_RECONFIG IP core. The reconfiguration signal on the ALTPLL_RECONFIG IP core starts the register chain transaction to update all PLL reconfigurable coefficients.

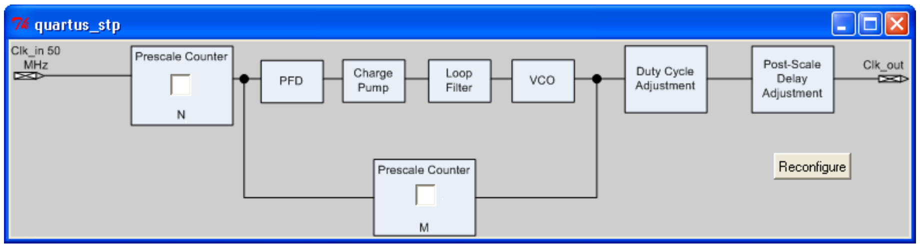
Figure 235. Block Diagram of Dynamic PLL Reconfiguration Design Example



This design example was created using a Nios II Development Kit, Stratix Edition. The file `sourceprobe_DE_dynamic_pll.zip` contains all the necessary files for running this design example, including the following:

- `Readme.txt`—A text file that describes the files contained in the design example and provides instructions about running the Tk GUI shown in the figure below.
- `Interactive_Reconfig.qar`—The archived Intel Quartus Prime project for this design example.

Figure 236. Interactive PLL Reconfiguration GUI Created with Tk and In-System Sources and Probes Tcl Package



Related Links

- On-chip Debugging Design Examples
  to download the In-System Sources and Probes Example
- On-chip Debugging Design Examples
  to download the In-System Sources and Probes Example

19.7 Document Revision History

Table 132. Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
June 2014	14.0.0	Updated formatting.
June 2012	12.0.0	Removed survey link.
November 2011	10.1.1	Template update.
December 2010	10.1.0	Minor corrections. Changed to new document template.
July 2010	10.0.0	Minor corrections.
November 2009	9.1.0	• Removed references to obsolete devices. • Style changes.
March 2009	9.0.0	No change to content.
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	8.0.0	• Documented that this feature does not support simulation on page 17–5 • Updated Figure 17–8 for Interactive PLL reconfiguration manager • Added hyperlinks to referenced documents throughout the chapter • Minor editorial updates

Related Links

Documentation Archive
For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.

20 Programming Intel FPGA Devices

The Intel Quartus Prime Programmer allows you to program and configure Intel FPGA CPLD, FPGA, and configuration devices. After compiling your design, use the Intel Quartus Prime Programmer to program or configure your device, to test the functionality of the design on a circuit board.

20.1 Programming Flow

The main stages of programming your device are:

1. Compiling your design, such that the Intel Quartus Prime Assembler generates the programming or configuration file.
2. Converting the programming or configuration file to target your configuration device and, optionally, creating secondary programming files.

Table 133. Programming and Configuration File Format

File Format	FPGA	CPLD	Configuration Device	Serial Configuration Device
SRAM Object File (`.sof`)	Yes	—	—	—
Programmer Object File (`.pof`)	—	Yes	Yes	Yes
JEDEC JESD71 STAPL Format File (`.jam`)	Yes	Yes	Yes	—
Jam Byte Code File (`.jbc`)	Yes	Yes	Yes	—

3. Programming and configuring the FPGA, CPLD, or configuration device using the programming or configuration file with the Intel Quartus Prime Programmer.

20.1.1 Stand-Alone Intel Quartus Prime Programmer

Intel FPGA offers the free stand-alone Programmer, which has the same full functionality as the Intel Quartus Prime Programmer in the Intel Quartus Prime software. The stand-alone Programmer is useful when programming your devices with another workstation, so you do not need two full licenses. You can download the stand-alone Programmer from the Download Center on the Altera website.

Stand-Alone Programmer Memory Limitations

The stand-alone Programmer may use significant memory during the following operations:

- During auto-detect operations
- When the programming file is added to the flash
- During manual attachment of the flash into the Programmer window

The 32-bit stand-alone Programmer can only use a limited amount of memory when launched in 32-bit Windows. Note the following specific limitations of 32-bit stand-alone Programmer:

Table 134. Stand-Alone Programmer Memory Limitations

Application	Maximum Flash Device Size	Flash Device Operation Using PFL
32-bit Stand-Alone Programmer	Up to 512 Mb	Single Flash Device
64-bit Stand-Alone Programmer	Up to 2 Gb	Multiple Flash Device

The stand-alone Programmer supports combination and/or conversion of Intel Quartus Prime programming files using the **Convert Programming Files** dialog box. You can convert programming files, such as Mask Settings File (`.msf`), Partial-Mask SRAM Object File (`.pmsf`), SRAM Object Files (`.sof`), or Programmer Object Files (`.pof`) into other file formats that support device configuration schemes for Intel FPGA devices.

Note the following device-specific file conversion limitations with use of the 32-bit stand-alone Programmer:

Table 135. Stand-Alone Programmer File Conversion Limitations

Programming File Conversion	Device Support
32-bit Programming File Conversion	All Supported Intel FPGA Devices Except Intel Arria 10
64-bit Programming File Conversion	All Supported Intel FPGA Devices

Related Links

Download Center

20.1.2 Optional Programming or Configuration Files

The Intel Quartus Prime software can generate optional programming or configuration files in various formats that you can use with programming tools other than the Intel Quartus Prime Programmer. When you compile a design in the Intel Quartus Prime software, the Assembler automatically generates either a `.sof` or `.pof`. The Assembler also allows you to convert FPGA configuration files to programming files for configuration devices.

Related Links

AN 425: Using Command-Line Jam STAPL Solution for Device Programming

20.1.3 Secondary Programming Files

The Intel Quartus Prime software generates programming files in various formats for use with different programming tools.

Table 136. File Types Generated by the Intel Quartus Prime Software and Supported by the Intel Quartus Prime Programmer

File Type	Generated by the Intel Quartus Prime Software	Supported by the Intel Quartus Prime Programmer
`.sof`	Yes	Yes
`.pof`	Yes	Yes
`.jam`	Yes	Yes
`.jbc`	Yes	Yes
JTAG Indirect Configuration File (`.jic`)	Yes	Yes
Serial Vector Format File (`.svf`)	Yes	—
Hexadecimal (Intel-Format) Output File (`.hexout`)	Yes	—
Raw Binary File (`.rbf`)	Yes	Yes (11)
Raw Binary File for Partial Reconfiguration (`.rbf`)	Yes	Yes (12)
Tabular Text File (`.ttf`)	Yes	—
Raw Programming Data File (`.rpd`)	Yes	—

20.2 Intel Quartus Prime Programmer Window

The Intel Quartus Prime **Programmer** window allows you to:

- Add your programming and configuration files.
- Specify programming options and hardware.
- Start the programming or configuration of the device.

To open the **Programmer** window, click **Tools ➤ Programmer**. As you proceed through the programming flow, the Intel Quartus Prime **Message** window reports the status of each operation.

Related Links

Programmer Page (Options Dialog Box)
In Intel Quartus Prime Help

20.2.1 Editing the Details of an Unknown Device

When the Intel Quartus Prime Programmer automatically detects devices with shared JTAG IDs, the Programmer prompts you to specify the device in the JTAG chain. If the Programmer does not prompt you to specify the device, you must manually add each device in the JTAG chain to the Programmer, and define the instruction register length of each device.

(11) Raw Binary File (`.rbf`) is supported by the Intel Quartus Prime Programmer in Passive Serial (PS) configuration mode.

(12) Raw Binary File for Partial Reconfiguration (`.rbf`) is supported by the Intel Quartus Prime Programmer in JTAG debug mode.

To edit the details of an unknown device, follow these steps:

1. Double-click the unknown device listed under the device column.
2. Click **Edit**.
3. Change the device **Name**.
4. Specify the **Instruction register Length**.
5. Click **OK**.
6. Save the `.cdf` file.

20.2.2 Setting Up Your Hardware

Before you can program or configure your device, you must have the correct hardware setup. The Intel Quartus Prime Programmer provides the flexibility to choose a download cable or programming hardware.

20.2.3 Setting the JTAG Hardware

The JTAG server allows the Intel Quartus Prime Programmer to access the JTAG hardware. You can also access the JTAG download cable or programming hardware connected to a remote computer through the JTAG server of that computer. With the JTAG server, you can control the programming or configuration of devices from a single computer through other computers at remote locations. The JTAG server uses the TCP/IP communications protocol.

20.2.3.1 Running JTAG Daemon with Linux

The JTAGD daemon allows a remote machine to program or debug a board that is connected to a Linux host over the network. The JTAGD daemon also allows multiple programs to use JTAG resources at the same time. The JTAGD daemon is the Linux version of a JTAG server.

Run the JTAGD daemon to avoid:

- The JTAGD server from exiting after two minutes of idleness.
- The JTAGD server from not accepting connections from remote machines, which might lead to an intermittent failure.

To run JTAGD as a daemon, follow these steps:

1. Create an `/etc/jtagd` directory.
2. Set the permissions of this directory and the files in the directory to allow you to have the read/write access.
3. Run `jtagd` (with no arguments) from your `quartus/bin` directory.

The JTAGD daemon is now running and does not terminate when you log off.

20.2.4 Using the JTAG Chain Debugger Tool

The JTAG Chain Debugger tool allows you to test the JTAG chain integrity and detect intermittent failures of the JTAG chain. In addition, the tool allows you to shift in JTAG instructions and data through the JTAG interface, and step through the test access port (TAP) controller state machine for debugging purposes. You access the tool by clicking **Tools ➤ JTAG Chain Debugger** on the Intel Quartus Prime software.

20.3 Programming and Configuration Modes

The following table lists the programming and configuration modes supported by Intel FPGA devices.

Table 137. Programming and Configuration Modes

Configuration Mode Supported by the Intel Quartus Prime Programmer	FPGA	CPLD	Configuration Device	Serial Configuration Device
JTAG	Yes	Yes	Yes	—
Passive Serial (PS)	Yes	—	—	—
Active Serial (AS) Programming	—	—	—	Yes
Configuration via Protocol (CvP)	Yes	—	—	—
In-Socket Programming	—	Yes (except for MAX II CPLDs)	Yes	Yes

Related Links

Configuration via Protocol (CvP) Implementation in V-series Intel FPGAs Devices User Guide
Describes the CvP configuration mode.

20.4 Design Security Keys

The Intel Quartus Prime Programmer supports the generation of encryption key programming files and encrypted configuration files for Intel FPGAs that support the design security feature. You can also use the Intel Quartus Prime Programmer to program the encryption key into the FPGA.

Related Links

AN 556: Using the Design Security Features in Intel FPGAs

20.5 Convert Programming Files Dialog Box

The **Convert Programming Files** dialog box in the Programmer allows you to convert programming files from one file format to another. To access the **Convert Programming Files** dialog box, click **File ➤ Convert Programming Files...** on the Intel Quartus Prime software.

For example, to store the FPGA data in configuration devices, you can convert the `.sof` data to another format, such as `.pof`, `.hexout`, `.rbf`, `.rpd`, or `.jic`, and then program the configuration device.

You can also configure multiple devices with an external host, such as a microprocessor or CPLD. For example, you can combine multiple `.sof` files into one `.pof` file. To save time in subsequent conversions, click **Save Conversion Setup** to save your conversion specifications in a Conversion Setup File (`.cof`). Click **Open Conversion Setup Data** to load your `.cof` setup in the **Convert Programming Files** dialog box.

Example 39. Conversion Setup File Contents

```
<?xml version="1.0" encoding="US-ASCII" standalone="yes"?>
<cof>
    <output_filename>output_file.pof</output_filename>
    <n_pages>1</n_pages>
    <width>1</width>
    <mode>14</mode>
    <sof_data>
        <user_name>Page_0</user_name>
        <page_flags>1</page_flags>
        <bit0>
            <sof_filename>/users/jbrossar/template/output_files/
template_test.sof</sof_filename>
        </bit0>
    </sof_data>
    <version>7</version>
    <create_cvp_file>0</create_cvp_file>
    <create_hps_iocsr>0</create_hps_iocsr>
    <auto_create_rpd>0</auto_create_rpd>
    <options>
        <map_file>1</map_file>
    </options>
    <MAX10_device_options>
        <por>0</por>
        <io_pullup>1</io_pullup>
        <auto_reconfigure>1</auto_reconfigure>
        <isp_source>0</isp_source>
        <verify_protect>0</verify_protect>
        <epof>0</epof>
        <ufm_source>0</ufm_source>
    </MAX10_device_options>
    <advanced_options>
        <ignore_epcs_id_check>0</ignore_epcs_id_check>
        <ignore_condone_check>2</ignore_condone_check>
        <plc_adjustment>0</plc_adjustment>
        <post_chain_bitstream_pad_bytes>-1</post_chain_bitstream_pad_bytes>
        <post_device_bitstream_pad_bytes>-1</post_device_bitstream_pad_bytes>
        <bitslice_pre_padding>1</bitslice_pre_padding>
    </advanced_options>
</cof>
```

Related Links

Convert Programming Files Dialog Box
In Intel Quartus Prime Help

20.5.1 Debugging Your Configuration

Use the **Advanced** option in the **Convert Programming Files** dialog box to debug your configuration. You must choose the advanced settings that apply to your Intel FPGA device. You can direct the Intel Quartus Prime software to enable or disable an advanced option by turning the option on or off in the **Advanced Options** dialog box. When you change settings in the **Advanced Options** dialog box, the change affects `.pof`, `.jic`, `.rpd`, and `.rbf` files.

The following table lists the **Advanced Options** settings in more detail:

Table 138. Advanced Options Settings

Option Setting	Description
Disable EPCS ID check	FPGA skips the EPCS silicon ID verification. Default setting is unavailable (EPCS ID check is enabled).

continued...

Option Setting	Description
	Applies to the single- and multi-device AS configuration modes on all FPGA devices.
Disable AS mode CONF_DONE error check	FPGA skips the `CONF_DONE` error check. Default setting is unavailable (AS mode `CONF_DONE` error check is enabled). Applies to single- and multi-device (AS) configuration modes on all FPGA devices. The `CONF_DONE` error check is disabled by default for Stratix V, Arria V, and Cyclone V devices for AS-PS multi device configuration mode.
Program Length Count adjustment	Specifies the offset you can apply to the computed PLC of the entire bitstream. Default setting is 0. The value must be an integer. Applies to single- and multi-device (AS) configuration modes on all FPGA devices.
Post-chain bitstream pad bytes	Specifies the number of pad bytes appended to the end of an entire bitstream. Default value is set to 0 if the bitstream of the last device is uncompressed. Set to 2 if the bitstream of the last device is compressed.
Post-device bitstream pad bytes	Specifies the number of pad bytes appended to the end of the bitstream of a device. Default value is 0. No negative integer. Applies to all single-device configuration modes on all FPGA devices.
Bitslice padding value	Specifies the padding value used to prepare bitslice configuration bitstreams, such that all bitslice configuration chains simultaneously receive their final configuration data bit. Default value is 1. Valid setting is 0 or 1. Use only in 2, 4, and 8-bit PS configuration mode, when you use an EPC device with the decompression feature enabled. Applies to all FPGA devices that support enhanced configuration devices.

The following table lists the symptoms you may encounter if a configuration fails, and describes the advanced options you must use to debug your configuration.

Failure Symptoms	Disable EPCS ID Check	Disable AS Mode CONF_DONE Error Check	PLC Settings	Post-Chain Bitstream Pad Bytes	Post-Device Bitstream Pad Bytes	Bitslice Padding Value
Configuration failure occurs after a configuration cycle.	—	Yes	Yes	Yes (13)	Yes (14)	—
Decompression feature is enabled.	—	Yes	Yes	Yes (13)	Yes (14)	—
Encryption feature is enabled.	—	Yes	Yes	Yes (13)	Yes (14)	—

continued...

(13) Use only for multi-device chain

(14) Use only for single-device chain

Failure Symptoms	Disable EPCS ID Check	Disable AS Mode CONF_DONE Error Check	PLC Settings	Post-Chain Bitstream Pad Bytes	Post-Device Bitstream Pad Bytes	Bitslice Padding Value
`CONF_DONE` stays low after a configuration cycle.	—	Yes	Yes [15]	Yes [13]	Yes [14]	—
`CONF_DONE` goes high momentarily after a configuration cycle.	—	Yes	Yes [16]	—	—	—
FPGA does not enter user mode even though `CONF_DONE` goes high.	—	—	—	Yes [13]	Yes [14]	—
Configuration failure occurs at the beginning of a configuration cycle.	Yes	—	—	—	—	—
Newly introduced EPCS, such as EPCS128.	Yes	—	—	—	—	—
Failure in `.pof` generation for EPC device using Intel Quartus Prime Convert Programming File Utility when the decompression feature is enabled.	—	—	—	—	—	Yes

20.5.2 Converting Programming Files for Partial Reconfiguration

The **Convert Programming File** dialog box supports the following programming file generation and option for Partial Reconfiguration:

- Partial-Masked SRAM Object File (**.pmsf**) output file generation, with **.msf** and **.sof** as input files.
- **.rbf** for Partial Reconfiguration output file generation, with a **.pmsf** as the input file.

  Note: The **.rbf** for Partial Reconfiguration file is only for Partial Reconfiguration.
- Providing the **Enable decompression during Partial Reconfiguration** option to enable the option bit for bitstream decompression during Partial Reconfiguration, when converting a full design **.sof** to any supported file type.

Related Links

Design Planning for Partial Reconfiguration

(15) Start with positive offset to the PLC settings

(16) Start with negative offset to the PLC settings

20.5.2.1 Generating .pmsf using a .msf and a .sof

To generate the **.pmsf** in the **Convert Programming Files** dialog box, follow these steps:

1. In the **Convert Programming Files** dialog box, under the **Programming file type** field, select **Partial-Masked SRAM Object File (.pmsf)**.
2. In the **File name field**, specify the necessary output file name.
3. In the **Input files to convert** field, add necessary input files to convert. You can add only a **.msf** and **.sof**.
4. Click **Generate**.

20.5.2.2 Generating .rbf for Partial Reconfiguration Using a .pmsf

After generating the **.pmsf**, convert the **.pmsf** to a **.rbf** for Partial Reconfiguration in the **Convert Programming Files** dialog box.

To generate the **.rbf** for Partial Reconfiguration, follow these steps:

1. In the **Convert Programming Files** dialog box, in the **Programming file type** field, select **Raw Binary File for Partial Reconfiguration (.rbf)**.
2. In the **File name** field, specify the output file name.
3. In the **Input files to conver**t field, add input files to convert. You can add only a **.pmsf**.
4. After adding the **.pmsf**, select the **.pmsf** and click **Properties**. The **PMSF File Properties** dialog box appears.
5. Make your selection either by turning on or turning off the following options:
   - **Compression option**—This option enables compression on Partial Reconfiguration bitstream. If you turn on this option, then you must turn on the **Enable decompression during Partial Reconfiguration** option.
   - **Enable SCRUB mode option**—The default of this option is based on AND/OR mode. This option is valid only when Partial Reconfiguration masks in your design are not overlapped vertically. Otherwise, you cannot generate the **.rbf** for Partial Reconfiguration.
   - **Write memory contents option**—This option is a workaround for initialized RAM/ROM in a Partial Reconfiguration region.

   For more information about these option, refer to the Design Planning for Partial Reconfiguration.
6. Click **OK**.
7. Click **Generate**.

20.5.2.3 Enable Decompression during Partial Reconfiguration Option

You can turn on the **Enable decompression during Partial Reconfiguration** option in the **SOF File Properties: Bitstream Encryption** dialog box, which can be accessed from the **Convert Programming File** dialog box. This option is available when converting a `.sof` to any supported programming file types listed in Table 136 on page 446.

This option is hidden for other targeted devices that do not support Partial Reconfiguration. To view this option in the **SOF File Properties: Bitstream Encryption** dialog box, the `.sof` must be targeted on an Intel FPGA device that supports Partial Reconfiguration.

If you turn on the **Compression** option when generating the `.rbf` for Partial Reconfiguration, then you must turn on the **Enable decompression during Partial Reconfiguration** option.

20.6 Flash Loaders

Parallel and serial configuration devices do not support the JTAG interface. However, you can use a flash loader to program configuration devices in-system via the JTAG interface. You can use an FPGA as a bridge between the JTAG interface and the configuration device. The Intel Quartus Prime software supports parallel and serial flash loaders.

20.7 JTAG Debug Mode for Partial Reconfiguration

The JTAG debug mode allows you to configure partial reconfiguration bitstream through the JTAG interface. Use this feature to debug PR bitstream and eventually helping you in your PR design prototyping. This feature is available for internal and external host. Using the JTAG debug mode forces the Data Source Controller to be in x16 mode.

During JTAG debug operation, the JTAG command sent from the Intel Quartus Prime Programmer ignores and overrides most of the Partial Reconfiguration IP core interface signals (`clk`, `pr_start`, `double_pr`, `data[]`, `data_valid`, and `data_read`).

Note: The TCK is the main clock source for PR IP core during this operation.

You can view the status of Partial Reconfiguration operation in the messages box and the Progress bar in the Intel Quartus Prime Programmer. The PR_DONE, PR_ERROR, and CRC_ERROR signals will be monitored during PR operation and reported in the Messages box at the end of the operation.

The Intel Quartus Prime Programmer can detect the number of PR_DONE instruction(s) in plain or compressed PR bitstream and, therefore, can handle single or double PR cycle accordingly. However, only single PR cycle is supported for encrypted Partial Reconfiguration bitstream in JTAG debug mode (provided that the specified device is configured with the encrypted base bitstream which contains the PR IP core in the design).

Note: Configuring an incompatible PR bitstream to the specified device may corrupt your design, including the routing path and the PR IP core placed in the static region. When this issue occurs, the PR IP core stays in an undefined state, and the Intel Quartus Prime Programmer is unable to reset the IP core. As a result, the Intel Quartus Prime Programmer generates the following error when you try to configure a new PR bitstream:

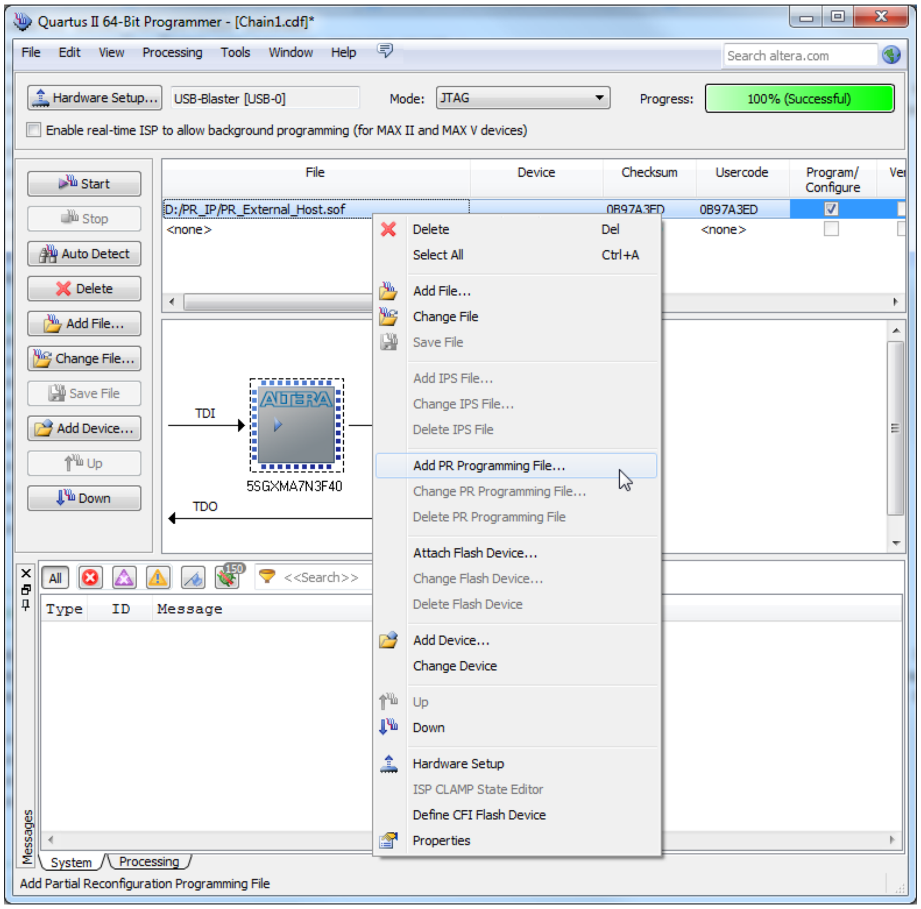
```
Error (12897): Partial Reconfiguration status: Can't reset the PR megafunction.
This issue occurred because the design was corrupted by an incompatible PR
bitstream in the previous PR operation. You must reconfigure the device with a
good design.
```

20.7.1 Configuring Partial Reconfiguration Bitstream in JTAG Debug Mode

To configure the Partial Reconfiguration bitstream in JTAG debug mode, follow these steps:

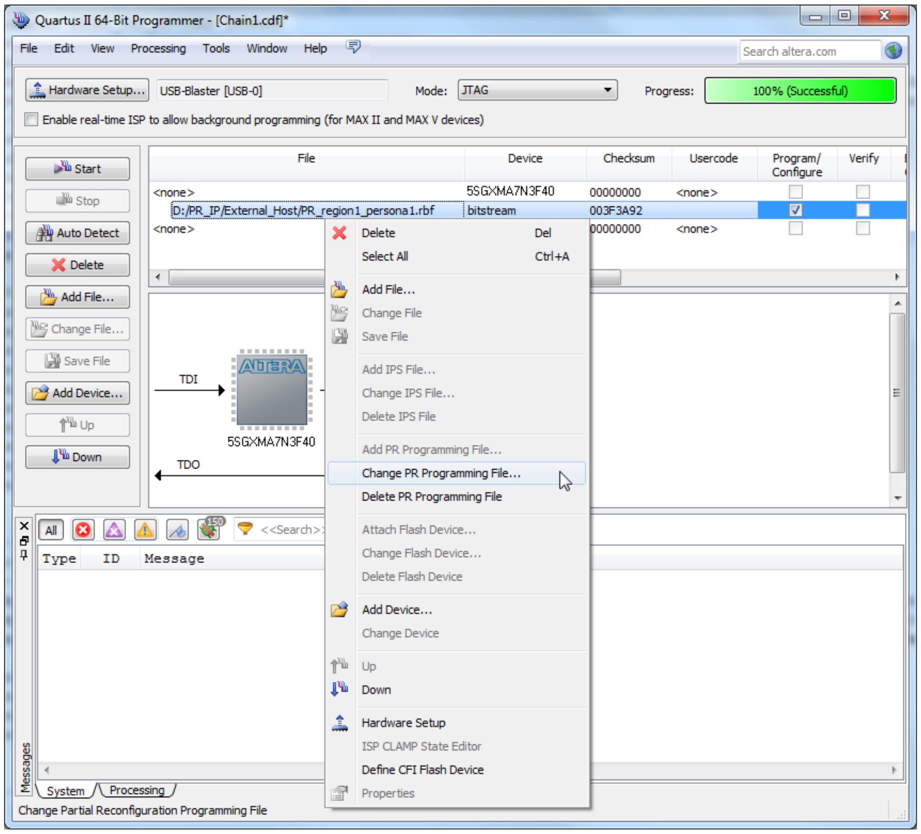
1. In the Intel Quartus Prime Programmer GUI, right click a highlighted base bitstream (in `.sof`) and then click **Add PR Programming File** to add the PR bitstream (`.rbf`).

Figure 237. Adding PR Programming File


Quartus II 64-Bit Programmer - [Chain1.cdf]*
File Edit View Processing Tools Window Help
Search altera.com
Hardware Setup... USB-Blaster [USB-0] Mode: JTAG Progress: 100% (Successful)
Enable real-time ISP to allow background programming (for MAX II and MAX V devices)
Start
Stop
Auto Detect
Delete
Add File...
Change File...
Save File
Add Device...
Up
Down
File Device Checksum Usercode Program/Configure
D:/PR_IP/PR_External_Host.sof 0B97A3ED 0B97A3ED
<none> <none>
TDI
TDO
5SGXMA7N3F40
Delete Del
Select All Ctrl+A
Add File...
Change File
Save File
Add IPS File...
Change IPS File...
Delete IPS File
Add PR Programming File...
Change PR Programming File...
Delete PR Programming File
Attach Flash Device...
Change Flash Device...
Delete Flash Device
Add Device...
Change Device
Up
Down
Hardware Setup
ISP CLAMP State Editor
Define CFI Flash Device
Properties
All <<Search>>
Type ID Message
System Processing
Messages
Add Partial Reconfiguration Programming File

2. After adding thePR bitstream, you can change or delete the Partial Reconfiguration programming file by clicking **Change PR Programming File** or **Delete PR Programming File**.

Figure 238. Change PR Programming File or Delete PR Programming File


Quartus II 64-Bit Programmer - [Chain1.cdf]*

Hardware Setup... USB-Blaster [USB-0] Mode: JTAG Progress: 100% (Successful)

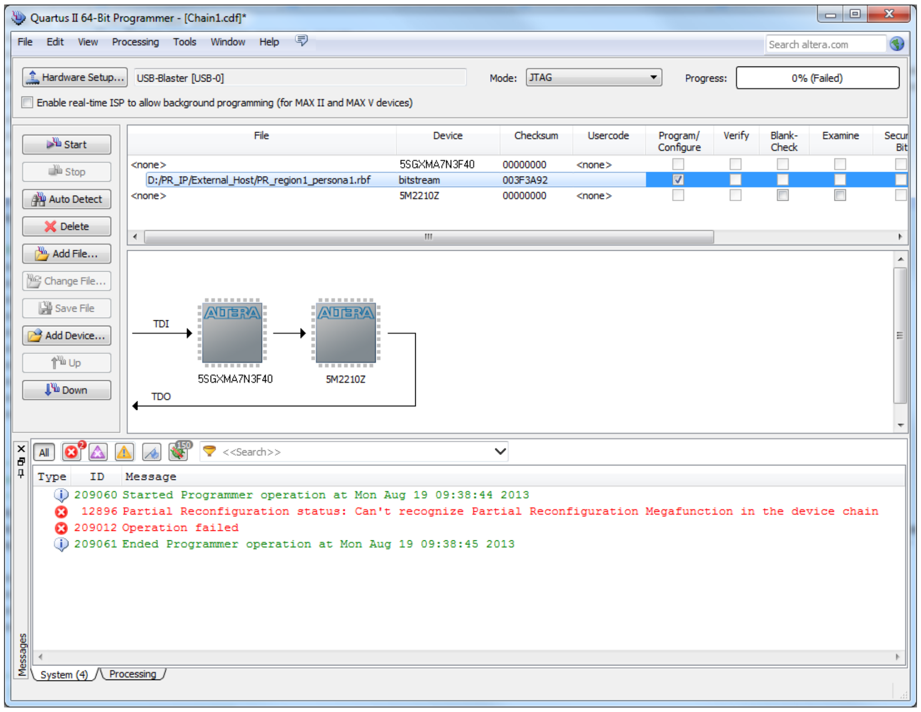
Enable real-time ISP to allow background programming (for MAX II and MAX V devices)

File	Device	Checksum	Usercode	Program/ Configure	Verify
<none>	5SGXMA7N3F40	00000000	<none>	☐	☐
D:/PR_IP/External_Host/PR_region1_persona1.rbf	bitstream	003F3A92		✓	☐
<none>		00000000	<none>	☐	☐

Change Partial Reconfiguration Programming File

3. Click **Start** to configure the PR bitstream. The Intel Quartus Prime Programmer generates an error message if the specified device does not contain the PR IP core in the design (you must instantiate the Partial Reconfiguration IP core in your design to use the JTAG debug mode).

Figure 239. Starting PR Bitstream Configuration



Quartus II 64-Bit Programmer - [Chain1.cdf]*

Hardware Setup... USB-Blaster [USB-0]   Mode: JTAG   Progress: 0% (Failed)

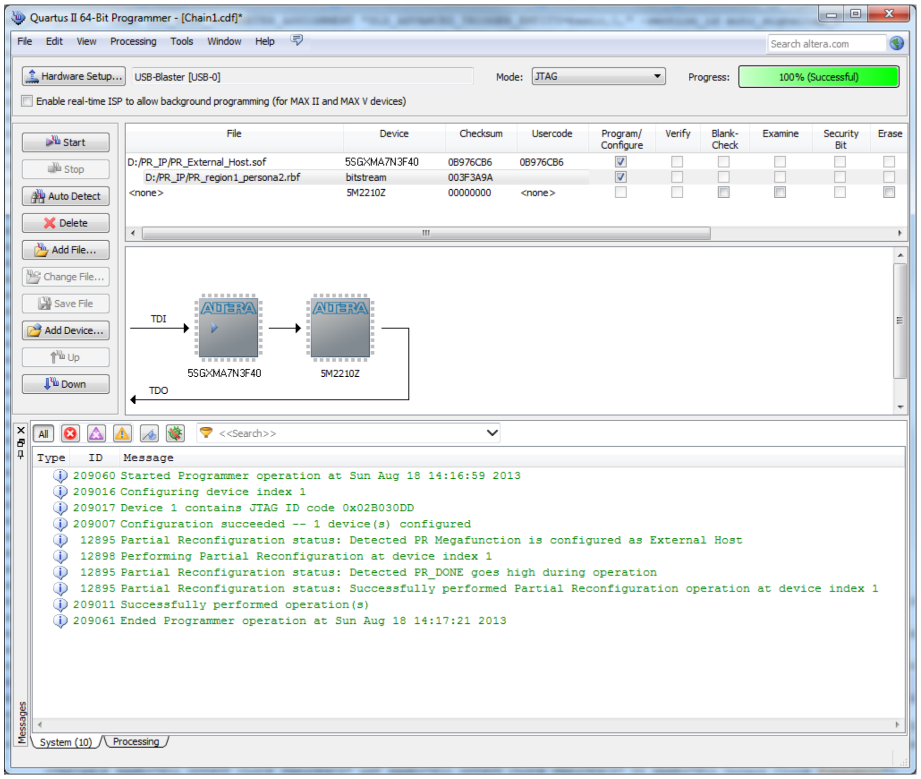
Enable real-time ISP to allow background programming (for MAX II and MAX V devices)

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine	Security Bit
<none>	5SGXMA7N3F40	00000000	<none>	☐	☐	☐	☐	☐
D:/PR_IP/External_Host/PR_region1_persona1.rbf	bitstream	003F3A92		✓	☐	☐	☐	☐
<none>	5M2210Z	00000000	<none>	☐	☐	☐	☐	☐

```
Type  ID     Message
      209060 Started Programmer operation at Mon Aug 19 09:38:44 2013
       12896 Partial Reconfiguration status: Can't recognize Partial Reconfiguration Megafunction in the device chain
      209012 Operation failed
      209061 Ended Programmer operation at Mon Aug 19 09:38:45 2013
```

4. Configure the valid `.rbf` in JTAG debug mode with the Intel Quartus Prime Programmer.

Figure 240. Configuring Valid .rbf



USB-Blaster [USB-0] Mode: JTAG Progress: 100% (Successful)

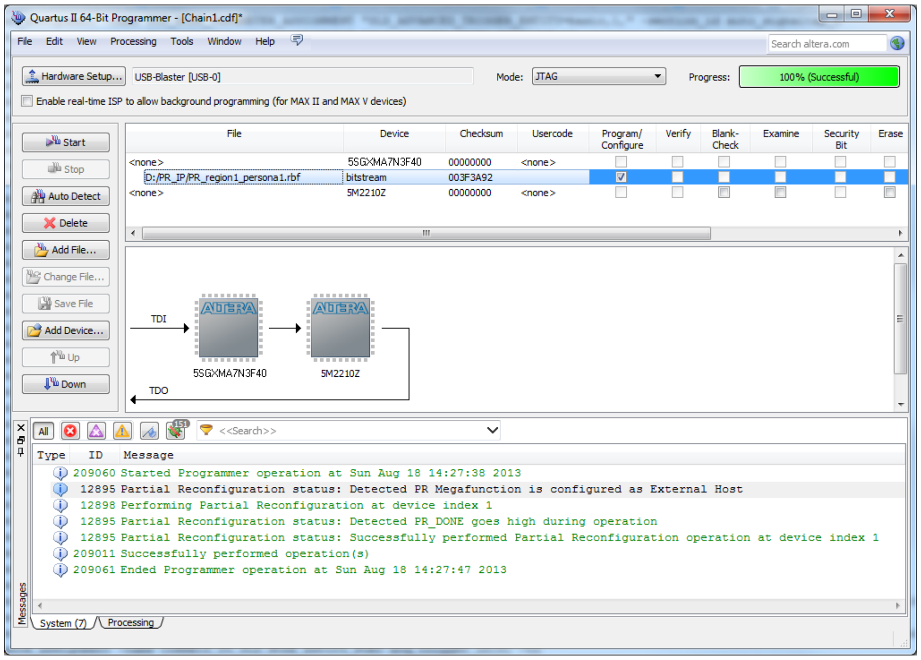
☐ Enable real-time ISP to allow background programming (for MAX II and MAX V devices)

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine	Security Bit	Erase
D:/PR_IP/PR_External_Host.sof	5SGXMA7N3F40	0B976CB6	0B976CB6	✓	☐	☐	☐	☐	☐
D:/PR_IP/PR_region1_persona2.rbf	bitstream	003F3A9A		✓	☐	☐	☐	☐	☐
<none>	5M2210Z	00000000	<none>	☐	☐	☐	☐	☐	☐

```
Type   ID     Message
       209060 Started Programmer operation at Sun Aug 18 14:16:59 2013
       209016 Configuring device index 1
       209017 Device 1 contains JTAG ID code 0x02B030DD
       209007 Configuration succeeded -- 1 device(s) configured
        12895 Partial Reconfiguration status: Detected PR Megafunction is configured as External Host
        12898 Performing Partial Reconfiguration at device index 1
        12895 Partial Reconfiguration status: Detected PR_DONE goes high during operation
        12895 Partial Reconfiguration status: Successfully performed Partial Reconfiguration operation at device index 1
       209011 Successfully performed operation(s)
       209061 Ended Programmer operation at Sun Aug 18 14:17:21 2013
```

5. The JTAG debug mode is also supported if the PR IP core is pre-programmed on the specified device.

Figure 241. Partial Reconfiguration IP Core Successfully Pre-programmed



Quartus II 64-Bit Programmer - [Chain1.cdf]*

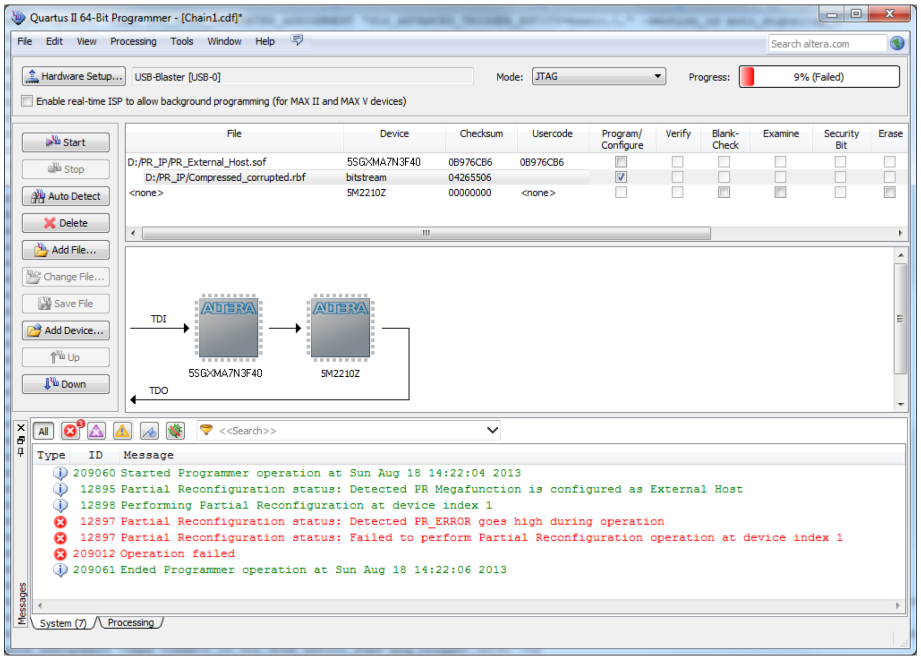
Hardware Setup... USB-Blaster [USB-0] Mode: JTAG Progress: 100% (Successful)

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine	Security Bit	Erase
<none>	5SGXMA7N3F40	00000000	<none>						
D:/PR_IP/PR_region1_persona1.rbf	bitstream	003F3A92		✓					
<none>	5M2210Z	00000000	<none>						

```
Type   ID    Message
     209060 Started Programmer operation at Sun Aug 18 14:27:38 2013
      12895 Partial Reconfiguration status: Detected PR Megafunction is configured as External Host
      12898 Performing Partial Reconfiguration at device index 1
      12895 Partial Reconfiguration status: Detected PR_DONE goes high during operation
      12895 Partial Reconfiguration status: Successfully performed Partial Reconfiguration operation at device index 1
     209011 Successfully performed operation(s)
     209061 Ended Programmer operation at Sun Aug 18 14:27:47 2013
```

6. The Intel Quartus Prime Programmer reports error when you try to configure the corrupted `.rbf` in JTAG debug mode.

Figure 242. Configuring Corrupted .rbf



Quartus II 64-Bit Programmer - [Chain1.cdf]*

Hardware Setup... USB-Blaster [USB-0] Mode: JTAG Progress: 9% (Failed)

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine	Security Bit	Erase
D:/PR_IP/PR_External_Host.sof	5SGXMA7N3F40	0B976CB6	0B976CB6						
D:/PR_IP/Compressed_corrupted.rbf	bitstream	04265506		✓					
<none>	5M2210Z	00000000	<none>						

```
Type   ID    Message
     209060 Started Programmer operation at Sun Aug 18 14:22:04 2013
      12895 Partial Reconfiguration status: Detected PR Megafunction is configured as External Host
      12898 Performing Partial Reconfiguration at device index 1
      12897 Partial Reconfiguration status: Detected PR_ERROR goes high during operation
      12897 Partial Reconfiguration status: Failed to perform Partial Reconfiguration operation at device index 1
     209012 Operation failed
     209061 Ended Programmer operation at Sun Aug 18 14:22:06 2013
```

20.8 Verifying if Programming Files Correspond to a Compilation of the Same Source Files

Intel Quartus Prime programming files support the project hash property, which helps you determine if two or more programming files correspond to a compilation of the same set of source files.

During compilation, the Intel Quartus Prime software produces an unique project hash, and embeds this value in the programming files (`.sof`). The project hash is available for Arria V, Stratix V, Cyclone V, Intel MAX 10, and Intel Arria 10 device families.

The project hash does not change for different builds of the Intel Quartus Prime software, or when you install a software patch. However, if you upgrade any IP with a different build or patch, the project hash changes.

20.8.1 Obtaining Project Hash for Arria V, Stratix V, Cyclone V and Intel MAX 10 Devices

To obtain the project hash value of a `.sof` programming file for a design targeted to Arria V, Stratix V, Cyclone V, and Intel MAX 10 devices, use the following command, which dumps out metadata information that includes the project hash.

```
quartus_cpf --info <sof-file-name>
```

Example 40. Output of Project Hash Extraction

In this example, the programming file name is `cb_intosc.sof`.

```
File: cb_intosc.sof
        File CRC: 0x0000
        Creator: Quartus Prime Compiler
        Version 17.0.0 Internal Build 565 02/09/2017 SJ Standard Edition
        Comment: UNIX
        Device: 5SGSMD5K2F40
        Data checksum: 0x02534E5A
        JTAG usercode: 0x02534E5A
        Project Hash:  0x556e737065636966696564
```

20.8.2 Obtaining Project Hash for Intel Arria 10 Devices

To obtain the project hash value of a `.sof` programming file for a design targeted to Intel Arria 10 devices, use the `quartus_asm` command-line executable (`quartus_asm.exe` in Windows) with the `--project_hash` option.

```
quartus_asm --project_hash <sof-file>
```

Example 41. Output of Project Hash Command

In this example, the programming file is `one_wire.sof`.

```
Info: *******************************************************************
Info: Running Quartus Prime Assembler
Info: Version 17.1.0 Build 569 08/23/2017 SJ Standard Edition
Info: Copyright (C) 2017  Intel Corporation. All rights reserved.
Info: Your use of Intel Corporation's design tools, logic functions
Info: and other software and tools, and its AMPP partner logic
Info: functions, and any output files from any of the foregoing
```

```
Info: (including device programming or simulation files), and any
Info: associated documentation or information are expressly subject
Info: to the terms and conditions of the Intel Program License
Info: Subscription Agreement, the Intel Quartus Prime License Agreement,
Info: the Intel MegaCore Function License Agreement, or other
Info: applicable license agreement, including, without limitation,
Info: that your use is for the sole purpose of programming logic
Info: devices manufactured by Intel and sold by Intel or its
Info: authorized distributors.  Please refer to the applicable
Info: agreement for further details.
Info: Processing started: Fri Aug 25 18:22:53 2017
Info: Command: quartus_asm --project_hash one_wire.sof
Info: Quartus(args): one_wire.sof
Info: Using INI file /data/test_asm/dis_all/quartus.ini
0x0e43694a1ffaf5da6088f900ffb0f7b6
Info (23030): Evaluation of Tcl script /tools/quartuskit/17.1std/quartus/
common/tcl/apps/qasm/project_hash.tcl was successful
Info: Quartus Prime Assembler was successful. 0 errors, 0 warnings
Info: Peak virtual memory: 1123 megabytes
Info: Processing ended: Fri Aug 25 18:22:59 2017
Info: Elapsed time: 00:00:06
Info: Total CPU time (on all processors): 00:00:03
```

20.9 Scripting Support

In addition to the Intel Quartus Prime Programmer GUI, you can use the Intel Quartus Prime command-line executable `quartus_pgm.exe` (or `quartus_pgm` in Linux) to access programmer functionality from the command line and from scripts. The programmer accepts `.pof`, `.sof`, and `.jic` programming or configuration files and `.cdf` files.

The following example shows a command that programs a device:

```
quartus_pgm –c byteblasterII –m jtag –o bpv;design.pof ←
```

Where:

- `-c byteblasterII` specifies the Intel FPGA Parallel Port Cable download cable
- `-m jtag` specifies the JTAG programming mode
- `-o bpv` represents the blank-check, program, and verify operations
- `design.pof` represents the `.pof` used for the programming

The Programmer automatically executes the erase operation before programming the device.

For Linux terminal, use:

```
quartus_pgm –c byteblasterII –m jtag –o bpv\;design.pof
```

Related Links

About Intel Quartus Prime Scripting
In Intel Quartus Prime Help

20.9.1 The jtagconfig Debugging Tool

You can use the `jtagconfig` command-line utility to check the devices in a JTAG chain and the user-defined devices. The `jtagconfig` command-line utility is similar to the auto detect operation in the Intel Quartus Prime Programmer.

For more information about the `jtagconfig` utility, use the help available at the command prompt:

```
jtagconfig [–h | --help]
```

Note: The help switch does not reference the `-n` switch. The `jtagconfig -n` command shows each node for each JTAG device.

Related Links

Command-Line Scripting
In *Intel Quartus Prime Standard Edition Handbook Volume 2*

20.9.2 Generating a Partial-Mask SRAM Object File using a Mask Settings File and a SRAM Object File

You can generate a `.pmsf` with the `quartus_cpf` command by typing the following command:

```
quartus_cpf -p <pr_revision.msf> <pr_revision.sof> <new_filename.pmsf>
```

20.9.3 Generating Raw Binary File for Partial Reconfiguration using a .pmsf

You can generate a `.rbf` for Partial Reconfiguration with the `quartus_cpf` command by typing the following command:

```
quartus_cpf –o foo.txt –c <pr_revision.pmsf> <pr_revision.rbf>
```

Note: You must run this command in the same directory where the files are located.

20.10 Document Revision History

Table 139. Document Revision History

Date	Version	Changes
2017.11.06	17.1.0	• Updated Project Hash feature.
2017.05.08	17.0.0	• Added Project Hash feature.
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
2015.05.04	15.0.0	Added Conversion Setup File (.cof) description and example.
December 2014	14.1.0	Updated the Scripting Support section to include a Linux command to program a device.
June 2014	14.0.0	• Added Running JTAG Daemon. • Removed Cyclone III and Stratix III devices references. • Removed MegaWizard Plug-In Manager references. • Updated Secondary Programming Files section to add notes about the Intel Quartus Prime Programmer support for **.rbf** files.
November 2013	13.1.0	• Converted to DITA format. • Added JTAG Debug Mode for Partial Reconfiguration and Configuring Partial Reconfiguration Bitstream in JTAG Debug Mode sections.

continued...

Date	Version	Changes
November 2012	12.1.0	• Updated Table 18–3 on page 18–6, and Table 18–4 on page 18–8. • Added "Converting Programming Files for Partial Reconfiguration" on page 18–10, "Generating .pmsf using a .msf and a .sof" on page 18–10, "Generating .rbf for Partial Reconfiguration Using a .pmsf" on page 18–12, "Enable Decompression during Partial Reconfiguration Option" on page 18–14 • Updated "Scripting Support" on page 18–15.
June 2012	12.0.0	• Updated Table 18–5 on page 18–8. • Updated "Intel Quartus Prime Programmer GUI" on page 18–3.
November 2011	11.1.0	• Updated "Configuration Modes" on page 18–5. • Added "Optional Programming or Configuration Files" on page 18–6. • Updated Table 18–2 on page 18–5.
May 2011	11.0.0	• Added links to Intel Quartus Prime Help. • Updated "Hardware Setup" on page 21–4 and "JTAG Chain Debugger Tool" on page 21–4.
December 2010	10.1.0	• Changed to new document template. • Updated "JTAG Chain Debugger Example" on page 20–4. • Added links to Intel Quartus Prime Help. • Reorganized chapter.
July 2010	10.0.0	• Added links to Intel Quartus Prime Help. • Deleted screen shots.
November 2009	9.1.0	No change to content.
March 2009	9.0.0	• Added a row to Table 21–4. • Changed references from "JTAG Chain Debug" to "JTAG Chain Debugger". • Updated figures.

Related Links

Documentation Archive

For previous versions of the *Intel Quartus Prime Handbook*, search the documentation archives.