Quartus Prime Pro Edition Handbook Volume 3: Verification

Subscribe

Send Feedback

QPP5V3
2015.11.02

101 Innovation Drive
San Jose, CA 95134
www.altera.com

Contents

Simulating Altera Designs 1

2015.11.02

QPP5V3 Subscribe Send Feedback

This document describes simulating designs that target Altera devices. Simulation verifies design behavior before device programming. The 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.

Simulator Support

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

Table 1-1: Supported Simulators

Vendor	Simulator	Version	Platform
Aldec	Active-HDL	10.2 Update 2	Windows
Aldec	Riviera-PRO	2015.06	Windows, Linux
Cadence	Incisive Enterprise	14.2	Linux
Mentor Graphics	ModelSim-Altera (provided)	10.4b	Windows, Linux
Mentor Graphics	ModelSim PE	10.4b	Windows
Mentor Graphics	ModelSim SE	10.4b	Windows, Linux
Mentor Graphics	QuestaSim	10.4b	Windows, Linux
Synopsys	VCS/VCS MX	2014,12-SP1	Linux

Simulation Levels

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

Table 1-2: 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 Altera and other IP providers.	• Design source/testbench • Altera simulation libraries • Altera IP plain text or IEEE encrypted RTL models • IP simulation models • Altera IP functional simulation models • Altera IP bus functional models • Qsys-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 • Altera simulation libraries • Post-synthesis or post-fit functional netlist • Altera IP bus functional models
Gate-level timing	Simulation using a post-fit timing netlist, testing functional and timing performance. Supported only for the Stratix IV, Cyclone IV, and MAX 10 device families.	• Testbench • Altera simulation libraries • Post-fit timing netlist • Post-fit Standard Delay Output File (**.sdo**). Not supported for MAX 10 devices.

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 Stratix IV and Cyclone IV device families. Use TimeQuest static timing analysis rather than gate-level timing simulation.

HDL Support

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

Table 1-3: HDL Support

Language	Description
VHDL	• For VHDL RTL simulation, compile design files directly in your simulator. To use NativeLink automation, analyze and elaborate your design in the Quartus Prime software, and then use the NativeLink simulator scripts to compile the design files in your simulator. You must also compile simulation models from the Altera simulation libraries and simulation models for the IP cores in your design. Use the Simulation Library Compiler or NativeLink to compile simulation models. • 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 Altera simulation libraries. • IEEE 1364-2005 encrypted Verilog HDL simulation models are encrypted separately for each Altera-supported simulation vendor. If you want to simulate the model in a VHDL design, you need either a simulator that is capable of VHDL/Verilog HDL co-simulation, or any Mentor Graphics single language VHDL simulator.
Verilog HDL SystemVerilog	• For RTL simulation in Verilog HDL or SystemVerilog, compile your design files in your simulator. To use NativeLink automation, analyze and elaborate your design in the Quartus Prime software, and then use the NativeLink simulator scripts to compile your design files in your simulator. You must also compile simulation models from the Altera simulation libraries and simulation models for the IP cores in your design. Use the Simulation Library Compiler or NativeLink 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 Altera IP cores in your design. • Altera provides the entry-level ModelSim-Altera software, along with precompiled Altera simulation libraries, to simplify simulation of Altera designs. Starting in version 15.0, the ModelSim-Altera 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-Altera software.
Schematic	You must convert schematics to HDL format before simulation. You can use the converted VHDL or Verilog HDL files for RTL simulation.

Simulation Flows

The Quartus® Prime software supports various method for integrating your supported simulator into the design flow.

Table 1-4: Simulation Flows

Simulation Flow	Description
NativeLink flow	The NativeLink automated flow supports a variety of design flows. Do not use NativeLink if you require direct control over every aspect of simulation. • Use NativeLink to generate simulation scripts to compile your design and simulation libraries, and to automatically launch your simulator. • Specify your own compilation, elaboration, and simulation scripts for testbench and simulation model files that have not been analyzed by the Quartus Prime software. • Use NativeLink to supplement your scripts by automatically compiling design files, IP simulation model files, and Altera simulation library models. • To use NativeLink for Arria 10 devices and later, you must add to your project the **.qsys** file generated for IP or Qsys system. To use NativeLink for all other device families, you must add to your project the **.qip** and **.sip** files generated for IP or Qsys systems. • The Quartus Prime Pro Edition software does not support NativeLink RTL simulation
Custom flows	Custom flows support manual control of all aspects of simulation, including the following: • Manually compile and simulate testbench, design, IP, and simulation model libraries, or write scripts to automate compilation and simulation in your simulator. • Use the Simulation Library Compiler to compile simulation libraries for all Altera devices and supported third-party simulators and languages. Use the custom flow if you require any of the following: • Custom compilation commands for design, IP, or simulation library model files (for example, macros, debugging or optimization options, or other simulator-specific options). • Multi-pass simulation flows. • Flows that use dynamically generated simulation scripts.
Specialized flows	Altera supports specialized flows for various design variations, including the following: • For simulation of Altera example designs, refer to the documentation for the example design or to the IP core user guide. • For simulation of Qsys designs, refer to *Creating a System with Qsys*. • For simulation of designs that include the Nios II embedded processor, refer to *Simulating a Nios II Embedded Processor*.

Related Information

- IP User Guide Documentation
- Creating a System with Qsys
- Simulating a Nios II Embedded Processor

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 Altera simulation libraries, and any other model libraries required for your design.

Generating Simulation Scripts

You can use Altera-provided utilities to generate a combined simulation script for Altera IP cores in your design. You can source these IP simulation scripts in your top-level project script. You can modify and reuse these simulation scripts to fit your simulation requirements.

You can use the following script variables:

- `TOP_LEVEL_NAME`—The top-level entity of your simulation is often a testbench that instantiates your design, and then your design instantiates IP cores and/or Qsys systems. Set the value of `TOP_LEVEL_NAME` to the top-level entity.
- `QSYS_SIMDIR`—Specifies the top-level directory containing the simulation files.
- Other variables control the compilation, elaboration, and simulation process.

Generating Version-Independent IP and Qsys Simulation Scripts

The Quartus Prime software includes useful utilities that generate simulation scripts for each IP core or Qsys system in your design. You can use these utilities to produce a single simulation script that does not require manual update for upgrades to Quartus Prime software or IP versions.

This scripted method generates simulation scripts that support ModelSim-Altera, all supported versions of Questa-SIM, VCS, VCSMX, NCSim, and Aldec simulators. These generated scripts are not suitable for entire design simulation because they lack top-level design information. However, you can easily source the generated scripts from your top-level simulation script. You can incorporate templates from the generated scripts into a top-level script.

Use the `ip-setup-simulation` utility to find all Altera IP cores and Qsys systems in your project. Next, run the `ip-make-simscript` utility to generate a combined IP simulation script. The `ip-setup-simulation` utility also automates regeneration of a combined simulation script following upgrade of the software. If you use simulation scripts, run the `ip-setup-simulation` utility after upgrading software or IP core version.

Set appropriate variables in the script, or edit the variable assignment directly in the script. If the simulation script is a Tcl file that is sourced in the simulator, set the variables before sourcing the script. If the simulation script is a shell script, pass in the variables as command-line arguments to the shell script.

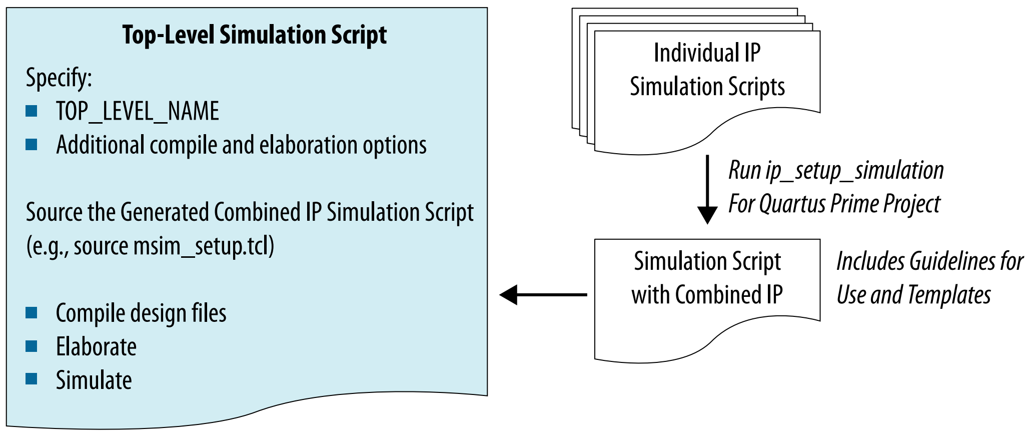
Table 1-5: IP Simulation Script Utilities

Utility	Description	Syntax	Generated Files
`ip-setup-simulation`	Finds all Altera IP cores in your project and automates regeneration of a combined simulation script after upgrading software or IP versions.	`ip-setup-simulation --quartus-project=<project>.qpf --output-directory=<directory>`	N/A
`ip-make-simscript`	Generates a single, combined simulation script for all of the IP cores specified on the command line. To use `ip-make-simscript`, specify one or more **.spd** files and an output directory in the command. Running the script compiles IP simulation models into various simulation libraries. Use the `compileto- work` option to compile all simulation files into a single work library. Use the `--use-relative-paths` option to use relative paths whenever possible	`ip-make-simscript --spd=<ipA.spd,ipB.spd> --output-directory=<directory>`	• Aldec—**aldec/rivierapro_setup.tcl** • Cadence—**cadence/ncsim_setup.sh** • Mentor Graphics—**mentor/msim_setup.tcl** • Synopsys—**synopsys/vcs/vcs_setup.sh**

Incorporating IP Simulation Scripts in Top-Level Scripts

You can incorporate generated IP core simulation scripts into a top-level simulation script that controls simulation of your entire design. After generating a combined IP simulation script, you can copy the template sections and modify them for use in a new top-level script file.

Figure 1-1: Incorporating IP Simulation Scripts into Top-Level Script


Top-Level Simulation Script
Specify:
TOP_LEVEL_NAME
Additional compile and elaboration options
Source the Generated Combined IP Simulation Script (e.g., source msim_setup.tcl)
Compile design files
Elaborate
Simulate
Individual IP Simulation Scripts
Run ip_setup_simulation For Quartus Prime Project
Simulation Script with Combined IP
Includes Guidelines for Use and Templates

1. Run `ip-setup-simulation` on the project:

```
ip-setup-simulation --quartus-project=<project>.qpf
    --output-directory=<directory>
```

2. Copy the template sections from the simulator-specific generated scripts and paste them into a new top-level file. The examples in this document assume that the top-level simulation script file is in the project directory, and that the generated simulation scripts are located in a directory one level below the project directory.
3. After pasting the template sections, remove the comments at the beginning of each line from the copied template sections.
4. Make any other modification to match your design simulation requirements, for example:
   a. Specify the `TOP_LEVEL_NAME` variable to the design's simulation top-level file.
   b. Compile the top-level HDL file (e.g. a test program) and all other files in the design.
   c. 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.

Incorporating Aldec IP Simulation Scripts

To incorporate generated Aldec simulation scripts into a top-level project simulation script, follow these steps:

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 user 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 user 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. Specify any other changes required to match your design simulation requirements.
5. Run the new top-level script from the generated simulation directory:

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

Incorporating Cadence IP Simulation Scripts

To incorporate generated Cadence IP simulation scripts into a top-level project simulation script, follow these steps:

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 user-defined 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 user-defined 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 \
```

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

```
ncvlog -sv "$QSYS_SIMDIR/../top.sv"
```

5. Specify any other changes required to match your design simulation requirements.
6. Run the resulting top-level script from the generated simulation directory by specifying the path to **ncsim.sh**.

Incorporating ModelSim IP Simulation Scripts

To incorporate generated ModelSim IP simulation scripts into a top-level project simulation script, follow these steps:

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 user 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 user 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. Specify any other changes required to match your design simulation requirements.
5. Run the resulting top-level script from the generated simulation directory:

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

Incorporating VCS IP Simulation Scripts

To incorporate generated Synopsys VCS simulation scripts into a top-level project simulation script, follow these steps:

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 user-defined 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 user-defined 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. Specify any other changes required to match your design simulation requirements.
6. Run the resulting top-level script from the generated simulation directory by specifying the path to **vcs_sim.sh**.

Incorporating VCS MX IP Simulation Scripts

To incorporate generated Synopsys VCS MX simulation scripts for use in top-level project simulation scripts, follow these steps:

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 user-defined 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 user-defined 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. Specify any other changes required to match your design simulation requirements.
6. Run the resulting top-level script from the generated simulation directory by specifying the path to **vcsmx_sim.sh**.

Compiling Simulation Models

The Quartus Prime software includes simulation models for all Altera IP cores. These models include IP functional simulation models, and device family-specific models in the **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 Altera simulation libraries using any of the following methods:

- Use the NativeLink feature to automatically compile your design, Altera IP, simulation model libraries, and testbench.
- Run the Simulation Library Compiler to compile all RTL and gate-level simulation model libraries for your device, simulator, and design language.
- Compile Altera simulation models manually with your simulator.

After you compile the simulation model libraries, you can reuse these libraries in subsequent simulations.

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

Related Information

Altera Simulation Models

Generating IP Simulation Files for RTL Simulation

The Quartus Prime software supports both Verilog HDL and VHDL simulation of encrypted and unencrypted Altera IP cores. If your design includes Altera IP cores, you must compile any corresponding IP simulation models in your simulator with the rest of your design and testbench. The Quartus Prime software generates and copies the simulation models for IP cores to your project directory.

You can use the following files to simulate your Altera IP variation.

Table 1-6: Altera IP Simulation Files

File Type	Description	File Name
Simulator setup script	Simulator-specific script to compile, elaborate, and simulate Altera IP models and simulation model library files. Copy the commands into your simulation script, or edit these files to compile, elaborate, and simulate your design and testbench.	Cadence • **cds.lib** • **ncsim_setup.sh** • **hdl.var** Mentor Graphics • **msim_setup.tcl** Synopsys • **synopsys_sim.setup** • **vcs_setup.sh** • **vcsmx_setup.sh** Aldec • **rivierapro_setup.tcl**
Simulation IP File (**.sip**) or Qsys System File (**.qsys**)	The **.sip** and **.qsys** files contain IP core simulation library mapping information. To use NativeLink for Arria 10 devices and later, you must add the **.qsys** file generated for IP or Qsys system to your project. To use NativeLink for all other device families, you must add the **.qip** and **.sip** files generated for IP or Qsys systems to your project.	**<*design name*>.sip**
IP functional simulation models	IP functional simulation models are cycle-accurate VHDL or Verilog HDL models generated by the Quartus Prime software for some Altera 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	Arria V, Cyclone V, Stratix V, and newer simulation model libraries and IP simulation models are provided in Verilog HDL and IEEE encrypted Verilog HDL. VHDL simulation of these models is supported using your simulator's co-simulation capabilities. IEEE encrypted Verilog HDL models are significantly faster than IP functional simulation models.	**<*my_ip*>.v**

Generating IP Functional Simulation Models for RTL Simulation

Altera provides IP functional simulation models for some Altera IP cores. To generate IP functional simulation models, follow these steps:

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

Note: Altera 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.

Note: Many recently released Altera IP cores support RTL simulation using IEEE Verilog HDL encryption. IEEE encrypted models are significantly faster than IP functional simulation models. You can simulate the models in both Verilog HDL and VHDL designs.

Related Information

AN 343: OpenCore Evaluation of AMPP Megafunctions

Running a Simulation (NativeLink Flow)

The NativeLink feature integrates your EDA simulator with the Quartus Prime software and automates the following simulation steps:

- Set and reuse simulation settings
- Generate simulator-specific files and simulation scripts
- Compile Altera simulation libraries
- Launch your simulator automatically following Quartus Prime Analysis & Elaboration, Analysis & Synthesis, or after a full compilation.

Note: To use NativeLink for Arria 10 devices and later, you must add to your project the **.qsys** file generated for IP or Qsys system. To use NativeLink for all other device families, you must add to your project the **.qip** and **.sip** files generated for IP or Qsys systems.

Setting Up Simulation (NativeLink Flow)

Before running simulation using the NativeLink flow, you must specify settings for your simulator in the Quartus Prime software. To specify simulation settings in the Quartus Prime software, follow these steps:

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

Table 1-7: Execution Paths for EDA Simulators

Simulator	Path
Mentor Graphics ModelSim-Altera	<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)

Simulator	Path
Aldec Active-HDL Aldec Riviera-PRO	<drive letter>**:**\<simlulator install path>**bin** (Windows) <simulator install path>/**bin** (Linux)

3. Click **Assignments** > **Settings** and specify options on the **Simulation** page and **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 Quartus Prime processing.
4. If your design includes a testbench, turn on **Compile test bench** and then click **Test Benches** to specify options for each testbench. Alternatively, turn on **Use script to compile testbench** and specify the script file.
5. If you want to use a script to setup simulation, turn on **Use script to setup simulation**.

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.

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.

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 Altera 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 Qsys simulation scripts—use the scripts generated for Altera IP cores and Qsys systems as templates to create simulation scripts. If your design includes multiple IP cores or Qsys 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. Altera recommends that you verify your design using RTL simulation for functionality and use the TimeQuest timing analyzer for timing. Timing simulation is not supported for Arria V, Cyclone V, Stratix V, and newer families.

Related Information

Running EDA Simulators

Document Revision History

This document has the following revision history.

Date	Version	Changes
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-Altera 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.
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.

Date	Version	Changes
November 2011	11.1.0	• Added information about encrypted Altera simulation model files. • Added information about IP simulation and NativeLink.

Related Information

Quartus Handbook Archive
For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.

Timing Analysis Overview 2

2015.11.02

QPP5V3 Subscribe Send Feedback

Timing Analysis Overview

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

The TimeQuest 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 TimeQuest analyzer.

Related Information

- **The Quartus Prime TimeQuest Timing Analyzer** on page 3-1
  For more information about the TimeQuest analyzer flow and TimeQuest examples.

TimeQuest Terminology and Concepts

Table 2-1: TimeQuest 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. (1)
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.

Term	Definition

Notes:

1. For Stratix® devices, the LUTs and registers are contained in logic elements (LE) and modeled as cells.

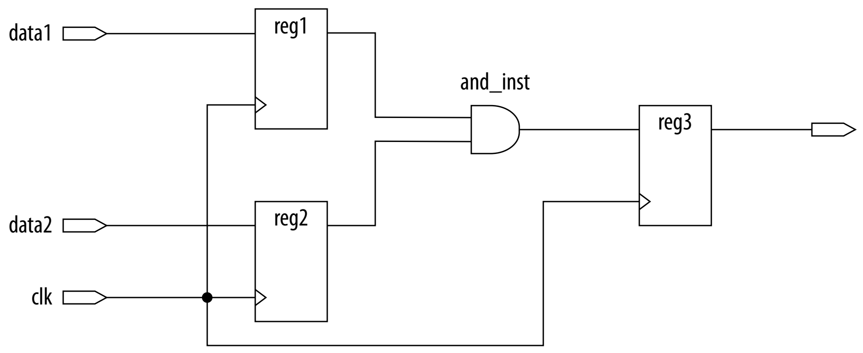
Timing Netlists and Timing Paths

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

The Timing Netlist

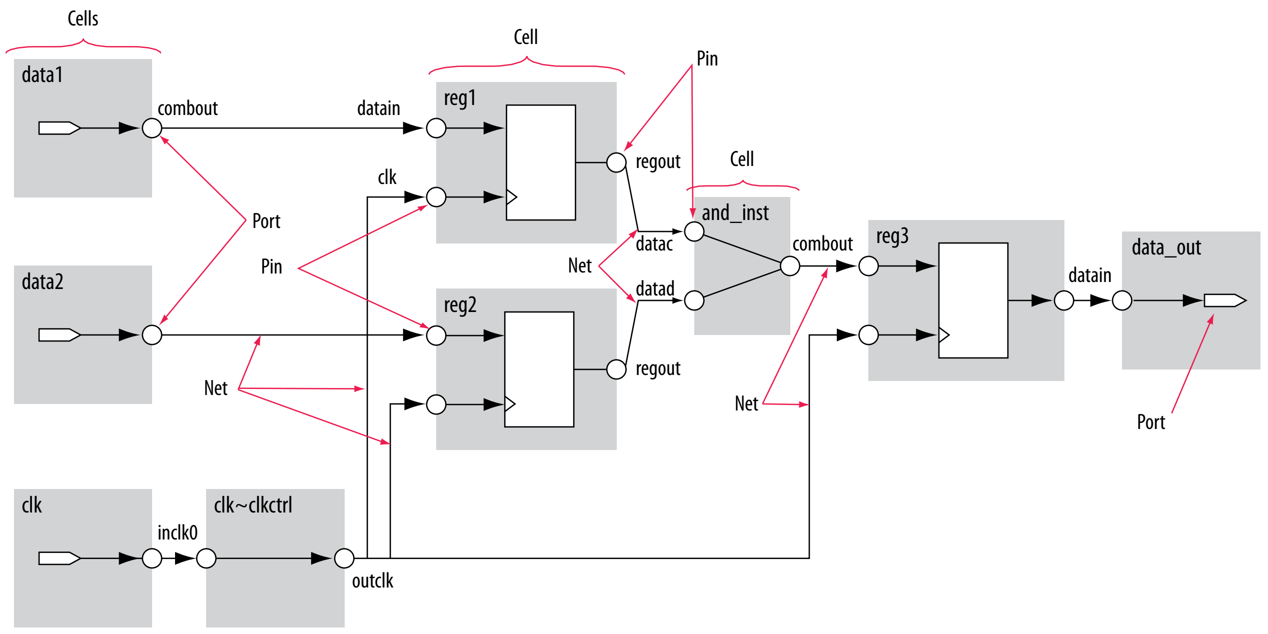
A sample design for which the TimeQuest analyzer generates a timing netlist equivalent.

Figure 2-1: Sample Design



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

Figure 2-2: The TimeQuest Analyzer Timing Netlist



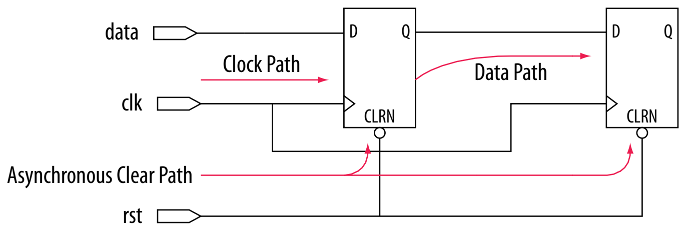
Timing Paths

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 TimeQuest 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 2-3: Path Types Commonly Analyzed by the TimeQuest Analyzer



In addition to identifying various paths in a design, the TimeQuest 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.

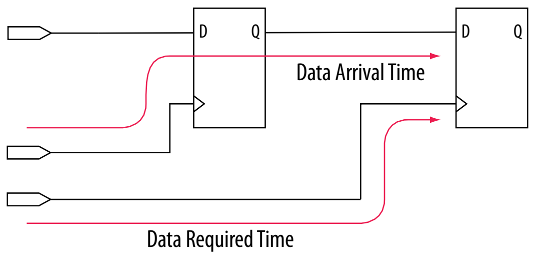
Data and Clock Arrival Times

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

The TimeQuest 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 TimeQuest 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 2-4: Data Arrival and Data Required Times



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

Figure 2-5: Data Arrival and Data Required Time Equations

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

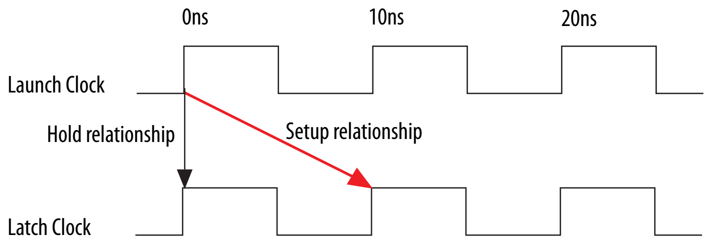
$$\text{Data Required Time} = \text{Latch Edge} + \text{Destination Clock Delay} - \mu t_{SU}$$

Launch and Latch Edges

All timing relies on one or more clocks. In addition to analyzing paths, the TimeQuest 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 2-6: Setup and Hold Relationship for Launch and Latch Edges 10ns Apart



In timing analysis, and with the TimeQuest 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.

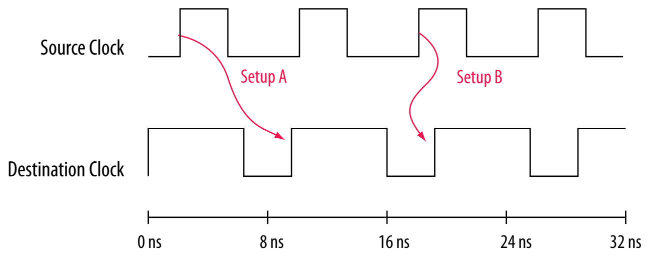
Note: If you do not constrain the clocks in your design, the Quartus Prime software analyzes in terms of a 1 GHz clock to maximize timing based Fitter effort. To ensure realistic slack values, you must constrain all clocks in your design with real values.

Clock Setup Check

To perform a clock setup check, the TimeQuest 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 TimeQuest 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 2-7: Setup Check



The TimeQuest 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 2-8: 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 TimeQuest analyzer performs setup checks using the maximum delay when calculating data arrival time, and minimum delay when calculating data required time.

Figure 2-9: 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 2-10: 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

Clock Hold Check

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

The TimeQuest 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 TimeQuest 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 TimeQuest 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 2-11: Setup and Hold Check Relationships

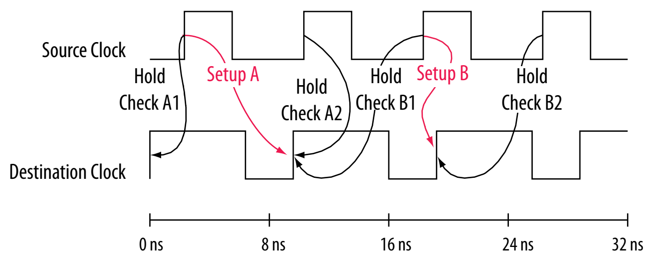


Figure 2-12: Clock Hold Slack for Internal Register-to-Register Paths

Clock Hold Slack = Data Arrival Time – Data Required 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_H + Hold Uncertainty

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

Figure 2-13: Clock Hold Slack Calculation from Input Port to Internal Register

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

Figure 2-14: Clock Hold Slack Calculation from Internal Register to Output Port

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

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 2-15: Recovery Slack Calculation if the Asynchronous Control Signal is Registered

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

Figure 2-16: Recovery Slack Calculation if the Asynchronous Control Signal is not Registered

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

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 TimeQuest 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 TimeQuest analyzer removal slack calculation is similar to the clock hold slack calculation, but it applies asynchronous control signals.

Figure 2-17: 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 2-18: 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

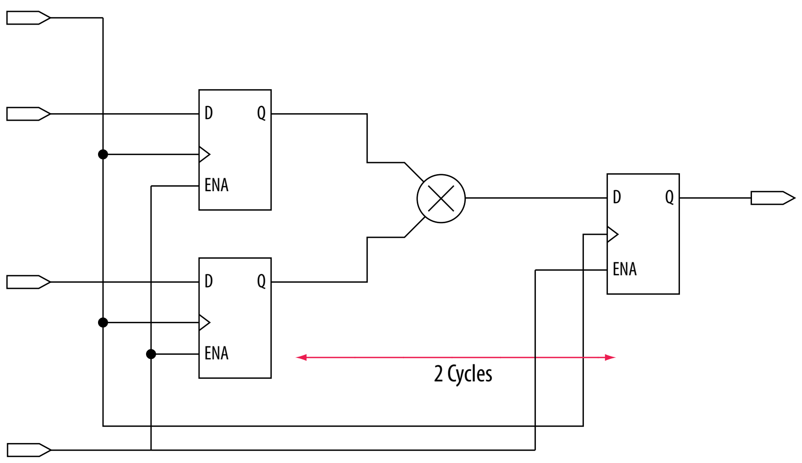
If the asynchronous reset signal is from a device pin, you must assign the **Input Minimum Delay** timing assignment to the asynchronous reset pin for the TimeQuest analyzer to perform removal analysis on the path.

Multicycle Paths

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

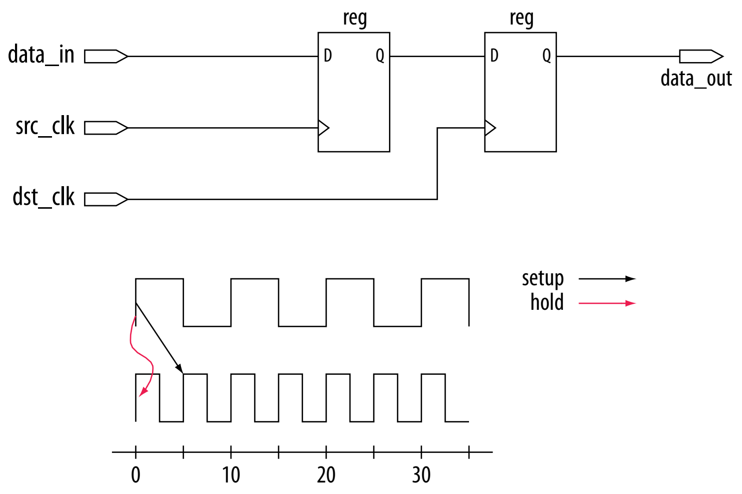
For example, a register may be required to capture data on every second or third rising clock edge. An example of a multicycle path between the input registers of a multiplier and an output register where the destination latches data on every other clock edge.

Figure 2-19: 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.

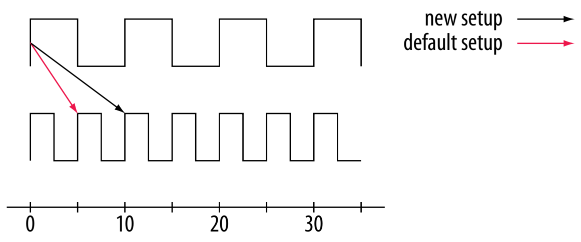
Figure 2-20: Register-to-Register Path and Default Setup and Hold Timing Diagram



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 2-21: Modified Setup Diagram



Related Information

- The Quartus Prime TimeQuest Timing Analyzer on page 3-1
  For more information about creating exceptions with multicycle paths.

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 TimeQuest 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 Quartus Prime software also protects these registers from optimizations that might negatively impact MTBF, such as register duplication and logic retiming. The Quartus Prime software can also optimize the MTBF of your design if the MTBF is too low.

Related Information

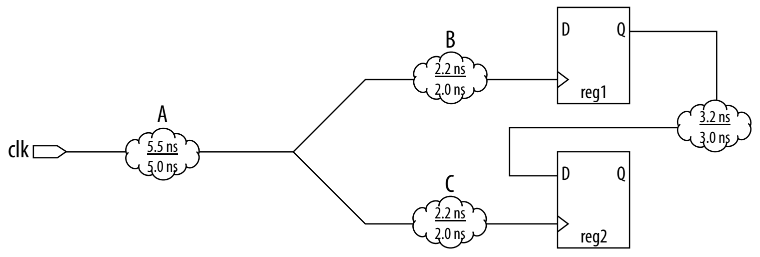
- **Understanding Metastability in FPGAs**
  For more information about metastability, its effects in FPGAs, and how MTBF is calculated.
- **Managing Metastability with the Quartus Prime Software**
  For more information about metastability analysis, reporting, and optimization features in the Quartus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

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 2-22: Typical Register to Register Path

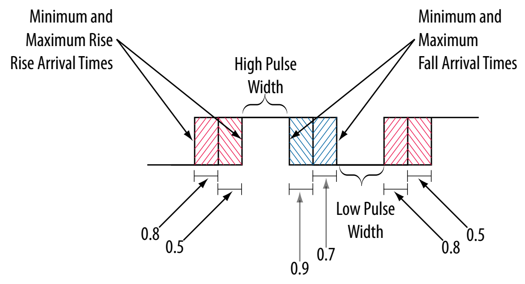


Segment A is the common clock path between `reg1` and `reg2`. The minimum delay is 5.0 ns; the maximum delay is 5.5 ns. The difference between the maximum and minimum delay value equals the common clock path pessimism removal value; in this case, the common clock path pessimism is 0.5 ns. The TimeQuest 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 TimeQuest analyzer subtracts the required minimum pulse width time from the actual minimum pulse width time. The TimeQuest 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 TimeQuest analyzer determines the required minimum pulse width time by the maximum rise, minimum rise, maximum fall, and minimum fall times.

Figure 2-23: 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 Information

TimeQuest Timing Analyzer Page (Settings Dialog Box)

For more information, refer to the Quartus Prime Help.

Clock-As-Data Analysis

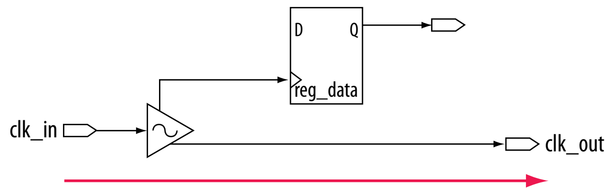
The majority of FPGA designs contain simple connections between any two nodes known as either a data path or a clock path.

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`.

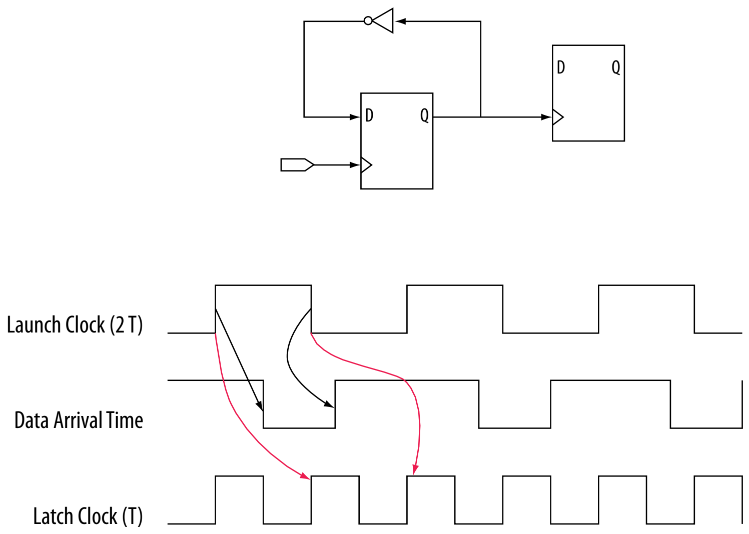
Figure 2-24: Simplified Source Synchronous Output



With clock-as-data analysis, the TimeQuest 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.

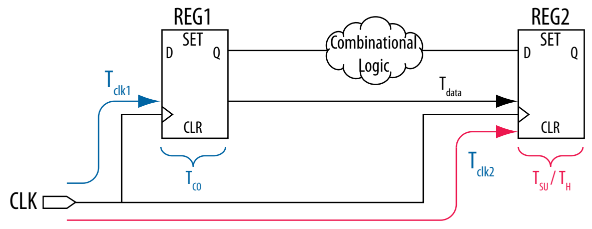
Figure 2-25: Clock Divider



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 2-26: Register-to-Register Path

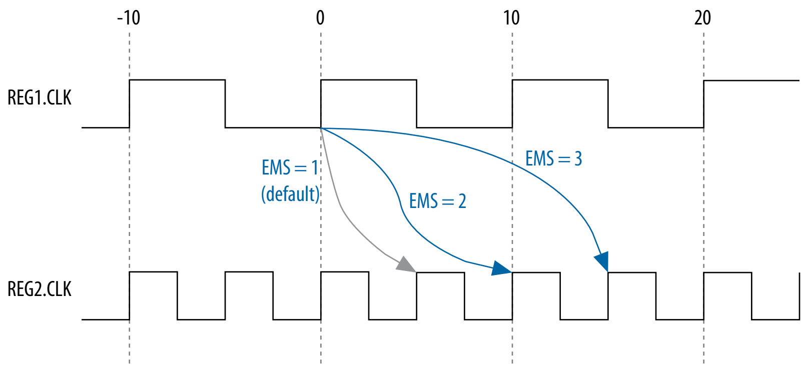


Multicycle Clock Setup

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

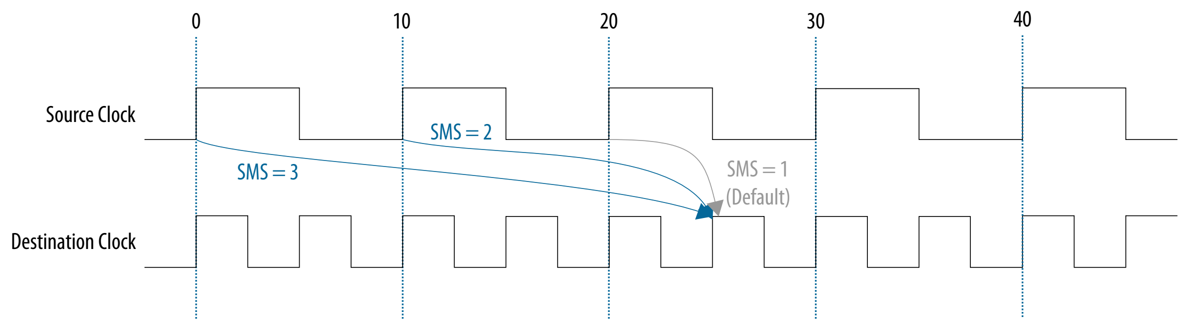
An end multicycle setup assignment modifies the latch edge of the destination clock by moving the latch edge the specified number of clock periods to the right of the determined default latch edge. The following figure shows various values of the end multicycle setup (EMS) assignment and the resulting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

Figure 2-27: End Multicycle Setup Values



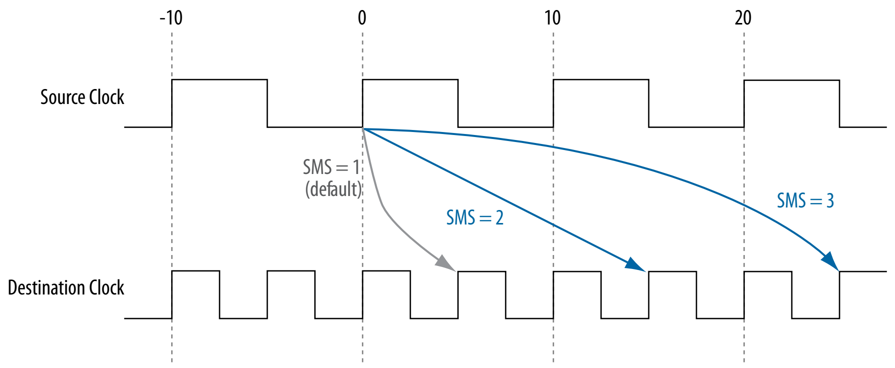
A start multicycle setup assignment modifies the launch edge of the source clock by moving the launch edge the specified number of clock periods to the left of the determined default launch edge. A start multicycle setup (SMS) assignment with various values can result in a specific launch edge.

Figure 2-28: Start Multicycle Setup Values



The setup relationship reported by the TimeQuest analyzer for the negative setup relationship.

Figure 2-29: Start Multicycle Setup Values Reported by the TimeQuest Analyzer



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 TimeQuest 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 TimeQuest 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 TimeQuest analyzer reports only the most restrictive hold check. The TimeQuest analyzer calculates the hold check by comparing launch and latch edges.

The calculation the TimeQuest analyzer performs to determine the hold check.

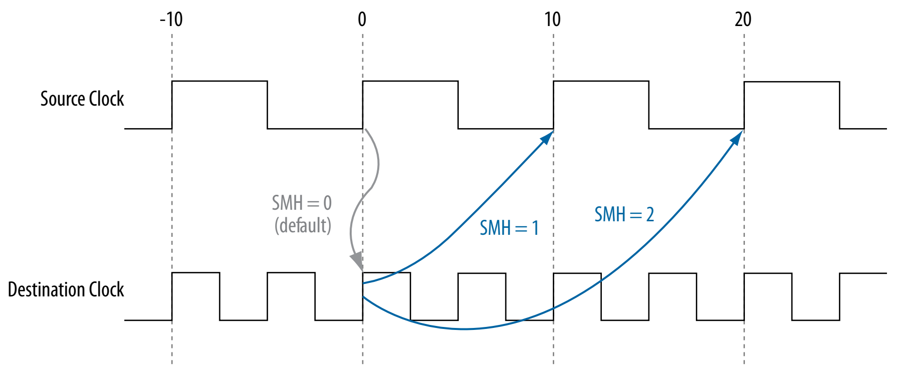
Figure 2-30: Hold Check

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

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

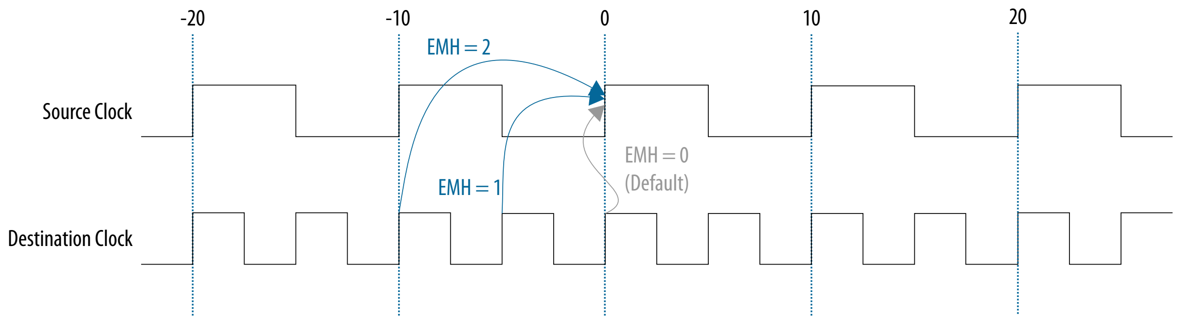
A start multicycle hold assignment modifies the launch edge of the destination clock by moving the latch edge the specified number of clock periods to the right of the determined default launch edge. he following figure shows various values of the start multicycle hold (SMH) assignment and the resulting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

Figure 2-31: Start Multicycle Hold Values



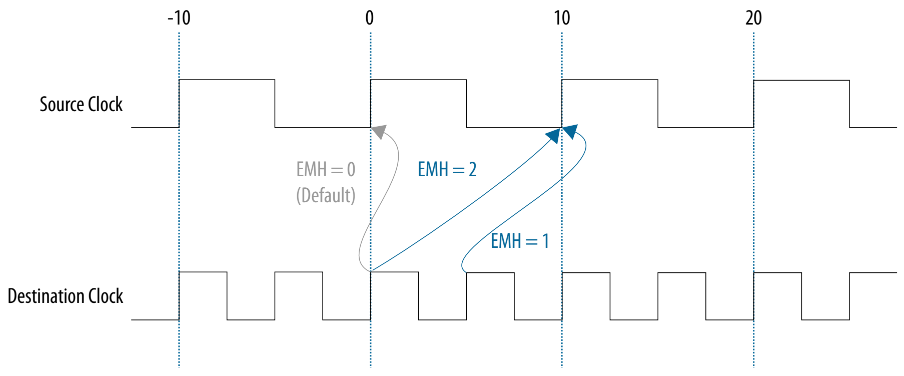
An end multicycle hold assignment modifies the latch edge of the destination clock by moving the latch edge the specific ed number of clock periods to the left of the determined default latch edge. he following figure shows various values of the end multicycle hold (EMH) assignment and the resulting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

Figure 2-32: End Multicycle Hold Values



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

Figure 2-33: End Multicycle Hold Values Reported by the TimeQuest Analyzer



Multicorner Analysis

The TimeQuest 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 2-2: 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 2-1: 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 Information

set_operating_conditions

get_available_operating_conditions
For more information about the `get_available_operating_conditions` command

Document Revision History

Table 2-3: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Quartus Prime*.
2014.12.15	14.1.0	Moved Multicycle Clock Setup Check and Hold Check Analysis section from the TimeQuest 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 Information

Quartus Handbook Archive

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.

The Quartus Prime TimeQuest Timing Analyzer 3

2015.11.02

QPP5V3 Subscribe Send Feedback

The Quartus Prime TimeQuest 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 TimeQuest 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 TimeQuest analyzer and timing analysis. The sections cover the following topics:

Related Information

- **Timing Analysis Overview** on page 2-1
  For more information about basic timing analysis concepts and how they pertain to the TimeQuest analyzer.
- **TimeQuest Timing Analyzer Resource Center**
  For more information about Altera resources available for the TimeQuest analyzer.
- **Altera Training**
  For more information about the TimeQuest analyzer.

Enhanced Timing Analysis for Arria 10

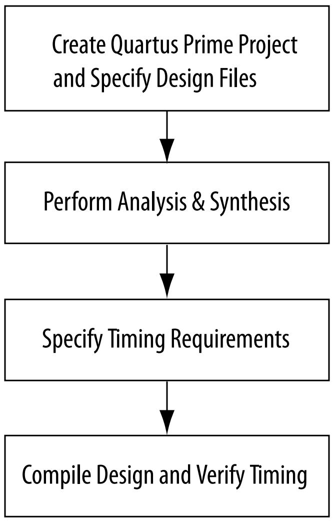
The TimeQuest Timing Analyzer supports new timing algorithms for the Arria® 10 device family which significantly improve the speed of the analysis.

These algorithms are enabled by default for 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 TimeQuest analysis analyzed the timing graph as many times as there were constraints in your Synopsys Design Constraint (SDC) file.

Recommended Flow for First Time Users

The Quartus Prime TimeQuest analyzer performs constraint validation to timing verification as part of the compilation flow. Both the TimeQuest 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 3-1: Design Flow with the TimeQuest Timing Analyzer


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

Creating and Setting Up your Design

You must first create your project in the 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 Altera 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 Quartus Prime software performs Fitting and Timing Analysis. Typically you must create an SDC file to constrain your design.

Related Information

SDC File Precedence on page 3-56

Specifying Timing Requirements

Before running timing analysis with the TimeQuest 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 TimeQuest Timing Analyzer Wizard to enter initial constraints for your design, and then refine timing constraints with the TimeQuest analyzer GUI.

Both the TimeQuest 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 TimeQuest analyzer, you can use template files included with the 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 Information

- Creating a Constraint File from Quartus Prime Templates with the Quartus Prime Text Editor on page 3-4
  For more information on using the <keyword keyref="qts-all" /> Text Editor templates for SDC constraints.
- Identifying the Quartus Prime Software Executable from the SDC File on page 3-69

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 Quartus Prime software populates an internal database with design element names. You must synthesize your design in order for the 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: When compiling for the 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
```

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

You can create an SDC file from constraint templates in the Quartus Prime software with the 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 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 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 Quartus Prime Text Editor, then click **Insert Template**TimeQuest.
   a. In the **Insert Template** dialog box, expand the **TimeQuest** 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 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 Information

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

Performing a Full Compilation

After creating initial timing constraints, compile your design.

During a full compilation, the Fitter uses the TimeQuest 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 Quartus Prime software.

If you are using the Quartus Prime Pro Edition software, you have some command line options available when performing iterative timing analysis on large designs.You can perform a less intensive analysis with `quartus_sta --mode=implement`. In this mode, the Quartus Prime software performs a reduced-corner timing analysis. When your modification iterations have achieved the desired result, you can use `quartus_sta --mode=finalize` to perform final Fitter optimizations and a full four-corner timing analysis.

Related Information

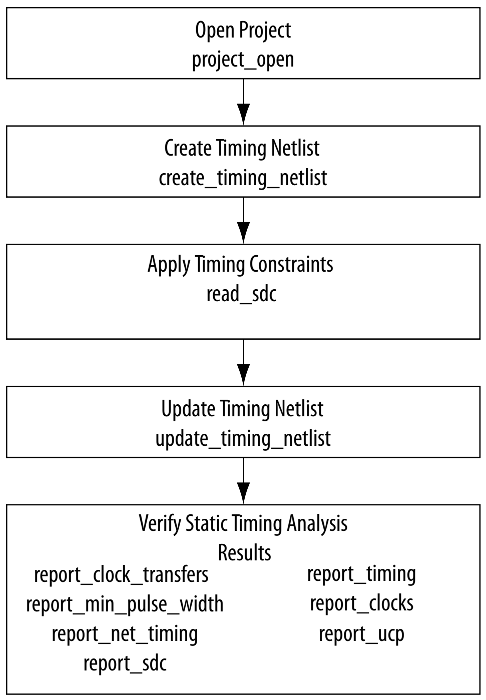
- **Analyzing Timing in Designs Compiled in Previous Versions** on page 3-6
  For more information about importing databases compiled in previous versions of the software.
- **Fitter Settings Page (Settings Dialog Box)**
  For more information about changing Fitter effort, refer to the Quartus Prime Help.

Verifying Timing

The TimeQuest 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 TimeQuest analyzer, and each part of the flow has a corresponding Tcl command.

Figure 3-2: The TimeQuest Timing Analyzer Flow



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 Quartus Prime software.

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

To export the database in the previous version of the 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 Quartus Prime software, click **File** > **Open** and select the Quartus Prime Project file (**.qpf**) for the project.

Once you import the database, you can perform any TimeQuest analyzer functions on the design without recompiling.

Timing Constraints

Timing analysis in the Quartus Prime software with the TimeQuest 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.

Recommended Starting SDC Constraints

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

create_clock on page 3-7

derive_pll_clocks on page 3-8

derive_clock_uncertainty on page 3-8

SDC Constraint Creation Summary on page 3-8
Those are the first three steps, which can usually be done very quickly. For a sample design with two clocks coming into it, your SDC file might look like this example:

set_clock_groups on page 3-9

Related Information

Creating a Constraint File from Quartus Prime Templates with the Quartus Prime Text Editor on page 3-4

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 TimeQuest, open the SDC file from TimeQuest or Quartus Prime, place the cursor where the new constraint will go, and go to **Edit** > **Insert Constraint**, and choose the constraint.

Warning: Using the **Constraints** menu option in the TimeQuest 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 TimeQuest, Altera recommends entering your constraints directly into your SDC with the **Edit** > **Insert Constraint** command.

Related Information

Creating Base Clocks on page 3-11

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 TimeQuest analyzer can automatically constrain them, wit thet `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 TimeQuest 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 Information

- Creating Base Clocks on page 3-11
- Deriving PLL Clocks on page 3-18

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 TimeQuest analyzer generates a warning if this command is not found in your SDC files.

Related Information

Accounting for Clock Effect Characteristics on page 3-21

SDC Constraint Creation Summary

Those are the first three steps, which can usually be done very quickly. For a sample design with two clocks coming into it, your SDC file might look like this example:

```
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
```

set_clock_groups

With the constraintsdiscusssed previously, most, if not all, of the clocks in the design are now constrained. In the TimeQuest 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 TimeQuest analyzer attempts to meet a 2ns setup relationship between these clocks unless you indicate that they are not related. The TimeQuest 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 TimeQuest 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 TimeQuest 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`. TimeQuest does not currently analyze crosstalk explicitly. Instead, the timing models use extra guard bands to account for any potential crosstalk-induced delays. TimeQuest 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 Information

- **Creating Generated Clocks** on page 3-15
- **Relaxing Setup with set_multicyle_path** on page 3-31
- **Accounting for a Phase Shift** on page 3-32

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 TimeQuest timing analyzer GUI.

In the **Task** panel of the **TimeQuest**analyzer, double-click on **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 TimeQuest 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 it will escape the whitespace, not 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 will 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 will have 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. TimeQuest 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.

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 TimeQuest analyzer supports SDC commands that accommodate various clocking schemes such as:

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

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 TimeQuest 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 TimeQuest 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 Information

- Accounting for Clock Effect Characteristics on page 3-21
- create_clock
- get_ports
  For more information about these commands, refer to Quartus Prime Help.

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 TimeQuest 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 Information

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

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.

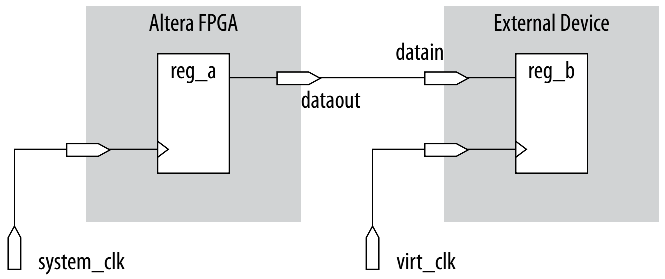
```
create_clock -period 10 -name my_virt_clk
```

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 3-3: 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 3-1: 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 Information

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

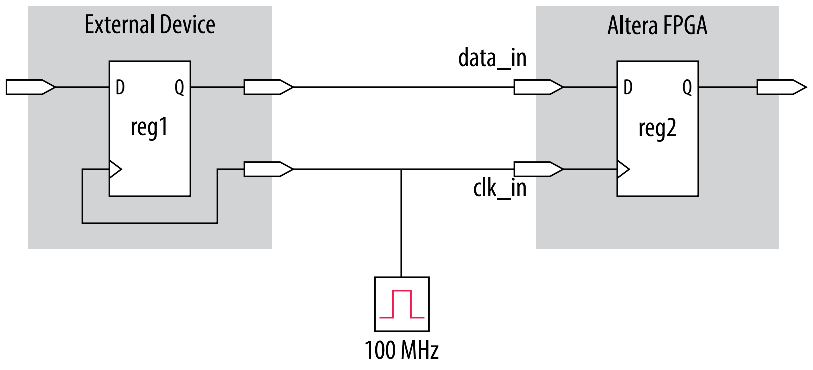
Example of Specifying an I/O Interface Clock

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 3-4: I/O Interface Clock Specifications



Example 3-2: 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]
```

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.

Related Information

Clock Uncertainty on page 3-21
For more information about clock uncertainty and clock transfers.

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.

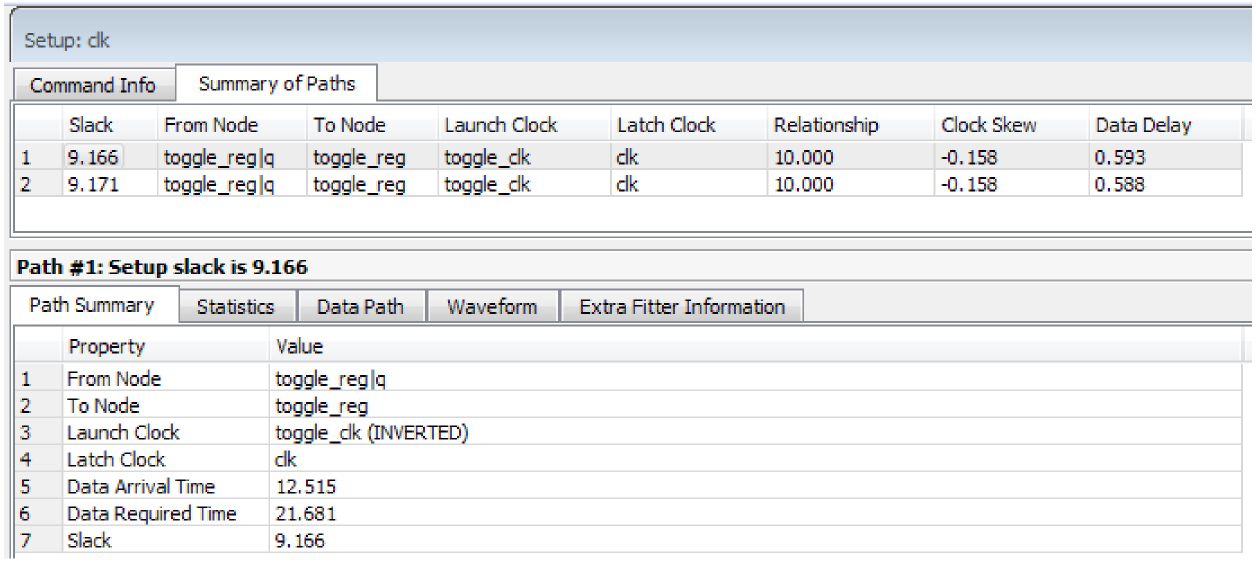
Use the `create_generated_clock` command to constrain generated clocks in your design.

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 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 TimeQuest. 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 TimeQuest 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 TimeQuest 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 Arria 10 device, the enhanced timing algorithms remove all common clock pessimism on paths treated as clock-as-data.

Figure 3-5: 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 TimeQuest 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 Information

- **Deriving PLL Clocks** on page 3-18
  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 Quartus Prime Help.
- **create_generate_clock**
- **derive_pll_clocks**
- **create_generated_clocks**
  For information about these commands, refer to Quartus Prime Help.

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 3-6: Clock Divider

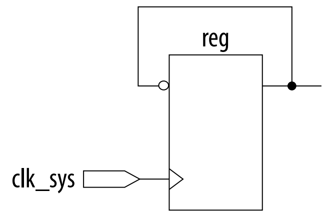
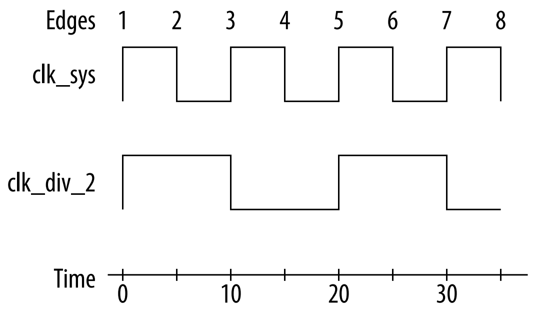


Figure 3-7: Clock Divider Waveform



```
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]
```

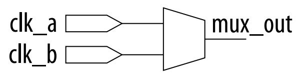
Or in order to have the clock source be the clock pin of the register you can use:

```
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]
```

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 TimeQuest analyzer does not analyze cross-domain paths between the generated clocks on the output of the clock mux.

Figure 3-8: 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
```

Deriving PLL Clocks

Use the `derive_pll_clocks` command to direct the TimeQuest 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

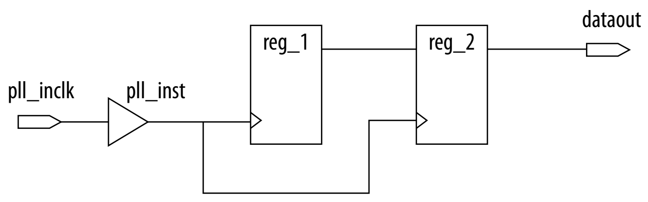
```
create_clock -period 10.0 -name fpga_sys_clk [get_ports fpga_sys_clk] \
    derive_pll_clocks
```

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 TimeQuest 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 3-9: Simple PLL Design



The TimeQuest 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 3-3: 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 Information

- **Creating Clock Groups** on page 3-19
  For more information about creating exclusive clock groups.
- **derive_pll_clocks**
- **Derive PLL Clocks**
  For more information about the derive_pll_clocks command.

Creating Clock Groups

The TimeQuest 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 TimeQuest 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 3-1: 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 3-2: 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 3-3: 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 3-4: 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 3-5: set_clock_groups -group {A C D}

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

Related Information

set_clock_groups

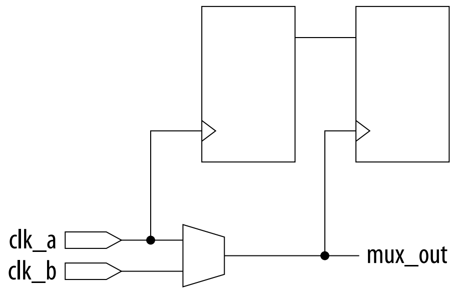
For more information about this command, refer to Quartus Prime Help.

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 3-10: 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 {port_a}]
set_clock_groups -exclusive -group {clk_a} -group {clk_b}
```

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 Information

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

Accounting for Clock Effect Characteristics

The clocks you create with the TimeQuest 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.

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 TimeQuest 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 Information

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

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 TimeQuest analyzer performs it anyway; it is a critical part of constraining your design correctly.

The TimeQuest 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 TimeQuest 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 TimeQuest 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 Information

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

Creating I/O Requirements

The TimeQuest analyzer reviews setup and hold relationships for designs in which an external source interacts with a register internal to the design. The TimeQuest 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.

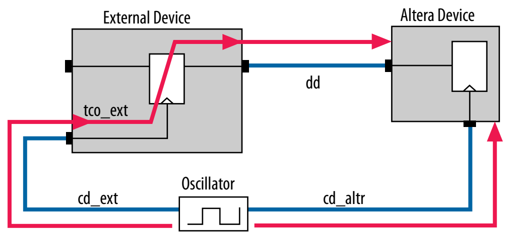
You must specify timing requirements, including internal and external timing requirements, before you fully analyze a design. With external timing requirements specified, the TimeQuest analyzer verifies the I/O interface, or periphery of the device, against any system specification.

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 TimeQuest analyzer to correctly derive clock uncertainties for interclock and intraclock transfers. The virtual clock defines the launching clock for the input port. The TimeQuest analyzer automatically determines the latching clock inside the device that captures the input data, because all clocks in the device are defined.

Figure 3-11: Input Delay



The calculation the TimeQuest analyzer performs to determine the typical input delay.

Figure 3-12: Input Delay Calculation

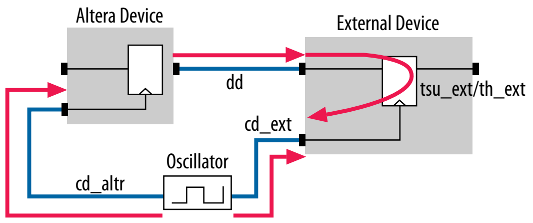
$$\text{input delay}_{MAX} = (\text{cd\_ext}_{MAX} - \text{cd\_altr}_{MIN}) + \text{tco\_ext}_{MAX} + \text{dd}_{MAX}$$
$$\text{input delay}_{MIN} = (\text{cd\_ext}_{MIN} - \text{cd\_altr}_{MAX}) + \text{tco\_ext}_{MIN} + \text{dd}_{MIN}$$

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 TimeQuest 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 3-13: Output Delay



The calculation the TimeQuest analyzer performs to determine the typical out delay.

Figure 3-14: 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 Information

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

Creating Delay and Skew Constraints

The TimeQuest analyzer supports the Synopsys Design Constraint format for constraining timing for the ports in your design. These constraints allow the TimeQuest 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.

Advanced I/O Timing and Board Trace Model Delay

The TimeQuest 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 3-4: Forcing Delay Annotation

```
create_timing_netlist -force_dat
```

Related Information

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

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 Arria 10 device, skew constraints are not affected by `set_clock_groups`.

Table 3-6: 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 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 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)

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

Arguments	Description
`-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`, `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 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, TimeQuest 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 TimeQuest 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 Information

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

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 3-7: 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
`-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, TimeQuest 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 will be ignored.

Related Information

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

Using create_timing_netlist

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

Your design should have a timing netlist before running the TimeQuest 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 **TimeQuest Timing Analyzer** page of the **Settings** dialog box.

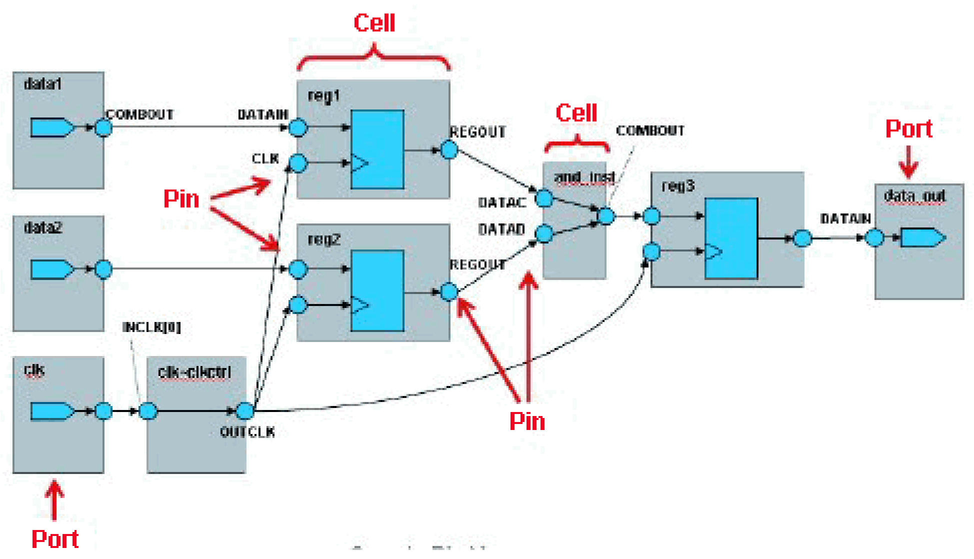
Note: The timing netlist created is based on the initial configuration of the design. Any configuration changes done by the design after the device enters user mode, for example, dynamic transceiver

[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.

reconfiguration, are not reflected in the timing netlist. This applies to all device families except transceivers on Arria 10 devices with the Multiple Reconfiguration Profiles feature.

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

Figure 3-15: How TimeQuest Interprets the Timing Netlist



Creating Timing Exceptions

Timing exceptions in the TimeQuest 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.

Precedence

If a conflict of node names occurs between 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. Finally, the remaining precedence for additional conflicts is order-dependent, such that the assignments most recently created overwrite, or partially overwrite, earlier assignments.

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 TimeQuest 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 Information

- Creating Clock Groups on page 3-19
  For more information about creating exclusive clock groups.
- set_false_path
  For more information about this command, refer to Quartus Prime Help.

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 TimeQuest 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 TimeQuest 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 Information

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

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 Information

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

Multicycle Paths

By default, the TimeQuest analyzer 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 TimeQuest analyzer does not report negative setup or hold relationships. When either a negative setup or a negative hold relationship is calculated, the TimeQuest 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 TimeQuest analyzer commands to modify either the launch or latch edge times that the uses to determine a setup relationship or hold relationship.

Table 3-8: 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

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.

Relaxing Setup with set_multicyle_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.

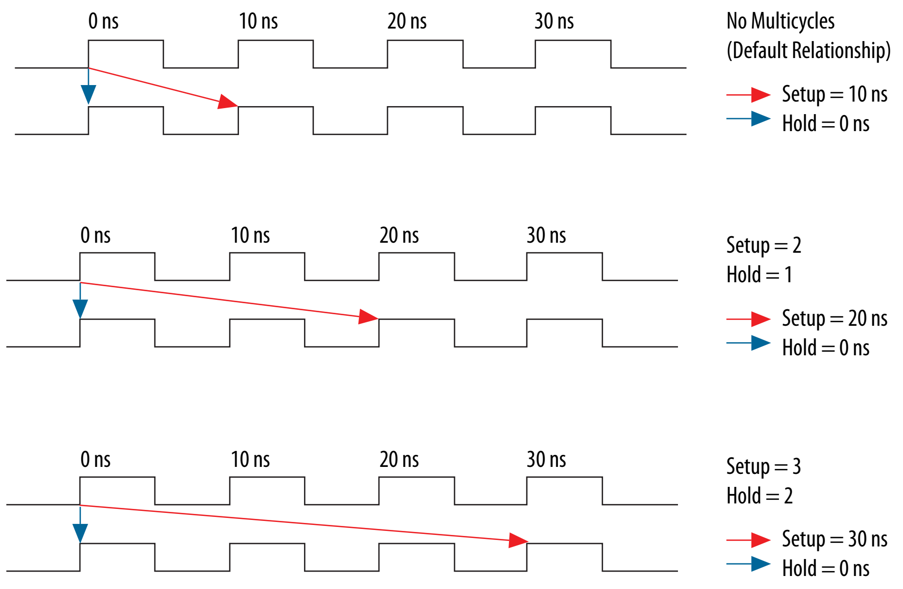
In this example, the source clock has a period of 10 ns, but a group of registers are enabled by a toggling clock, so they only toggle every other cycle. Since they are fed by a 10 ns clock, the TimeQuest analyzer reports a set up of 10 ns and a hold of 0 ns, However, since the data is transferring every 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 3-16: 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
```

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.

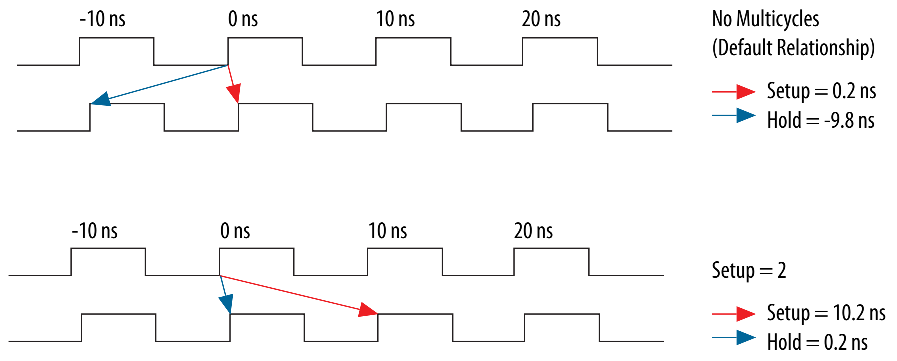
For example, the following code is a circumstance where a PLL phase-shifts one output forward by a small amount, in this case 0.2 ns.

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 3-17: 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 Information

- Same Frequency Clocks with Destination Clock Offset on page 3-41
  Refer to this topic for a more complete example.
- Same Frequency Clocks with Destination Clock Offset on page 3-41
  Refer to this topic for a more complete example.
- set_multicycle_path
  For more information about this command, refer to the Quartus Prime Help.

Examples of Basic Multicycle Exceptions

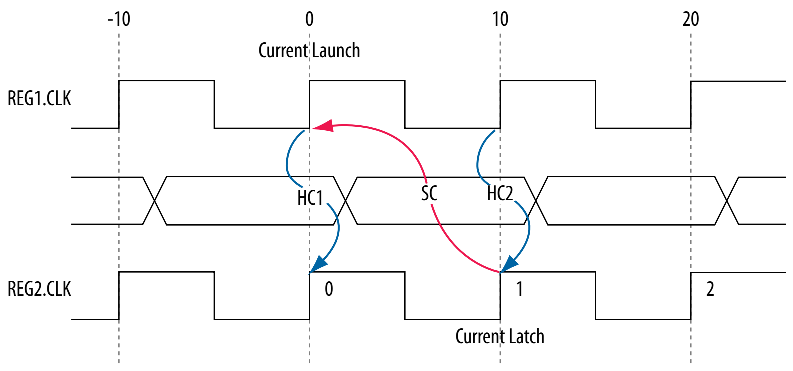
Each example explains how the multicycle exceptions affect the default setup and hold analysis in the TimeQuest analyzer. The multicycle exceptions are applied to a simple register-to-register circuit. Both the source and destination clocks are set to 10 ns.

Default Settings

By default, the TimeQuest analyzer performs a single-cycle analysis to determine the setup and hold checks. Also, by default, the TimeQuest 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 3-18: Default Timing Diagram



The calculation that the TimeQuest analyzer performs to determine the setup check.

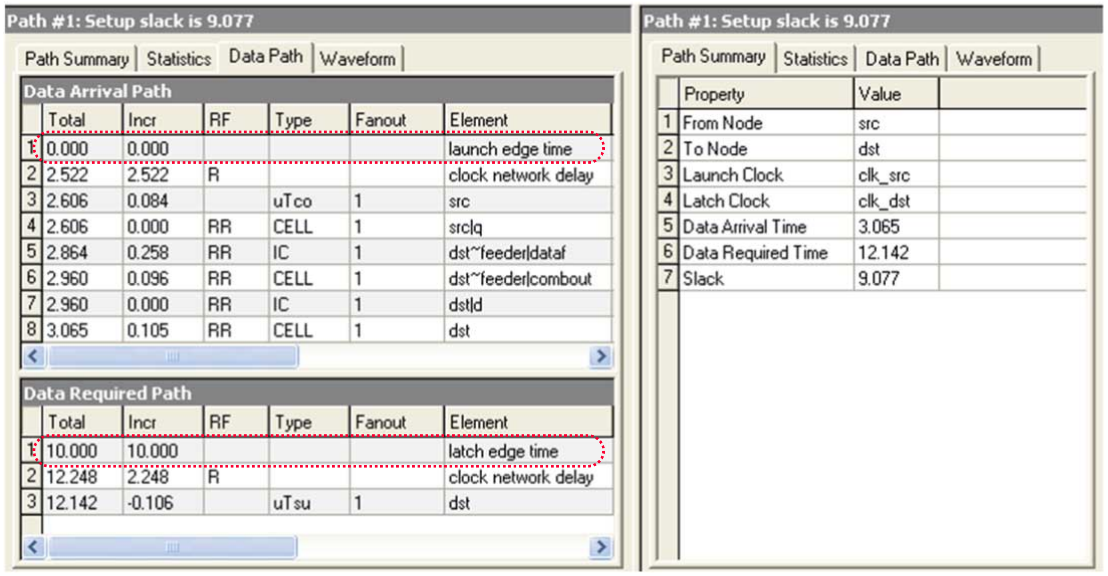
Figure 3-19: Setup Check

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

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 TimeQuest analyzer with the launch and latch edges highlighted.

Figure 3-20: Setup Report



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

	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 TimeQuest analyzer performs to determine the hold check. Both hold checks are equivalent.

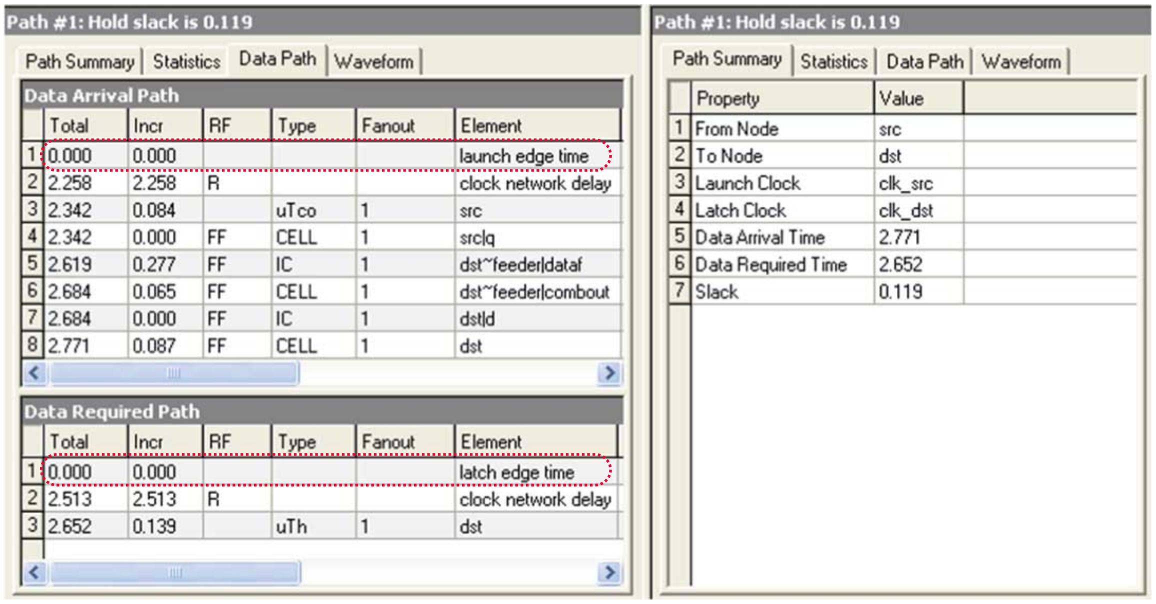
Figure 3-21: Hold Check

$$
\begin{aligned}
\text{hold check 1} &= \text{current launch edge} - \text{previous latch edge} \\
&= 0\text{ ns} - 0\text{ ns} \\
&= 0\text{ ns} \\
\\
\text{hold check 2} &= \text{next launch edge} - \text{current latch edge} \\
&= 10\text{ ns} - 10\text{ ns} \\
&= 0\text{ ns}
\end{aligned}
$$

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 TimeQuest analyzer with the launch and latch edges highlighted.

Figure 3-22: 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

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

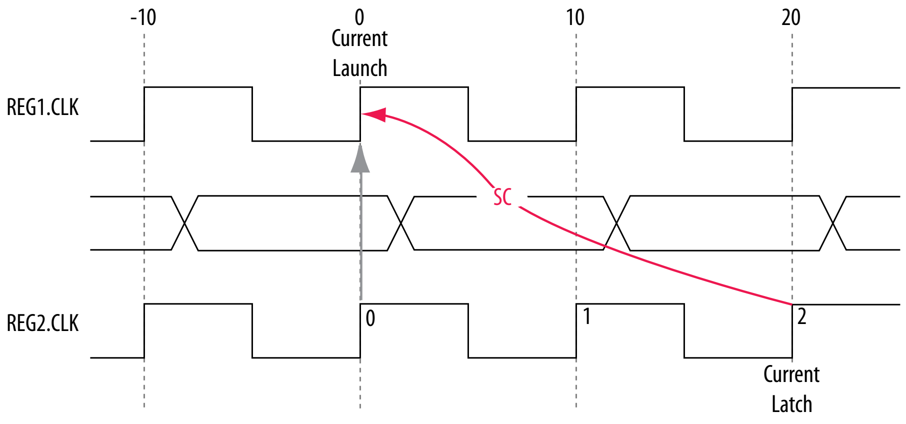
```
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] \
    -setup -end 2
```

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 TimeQuest analyzer performs. The latch edge is a clock cycle later than in the default single-cycle analysis.

Figure 3-23: Setup Timing Diagram



The calculation that the TimeQuest analyzer performs to determine the setup check.

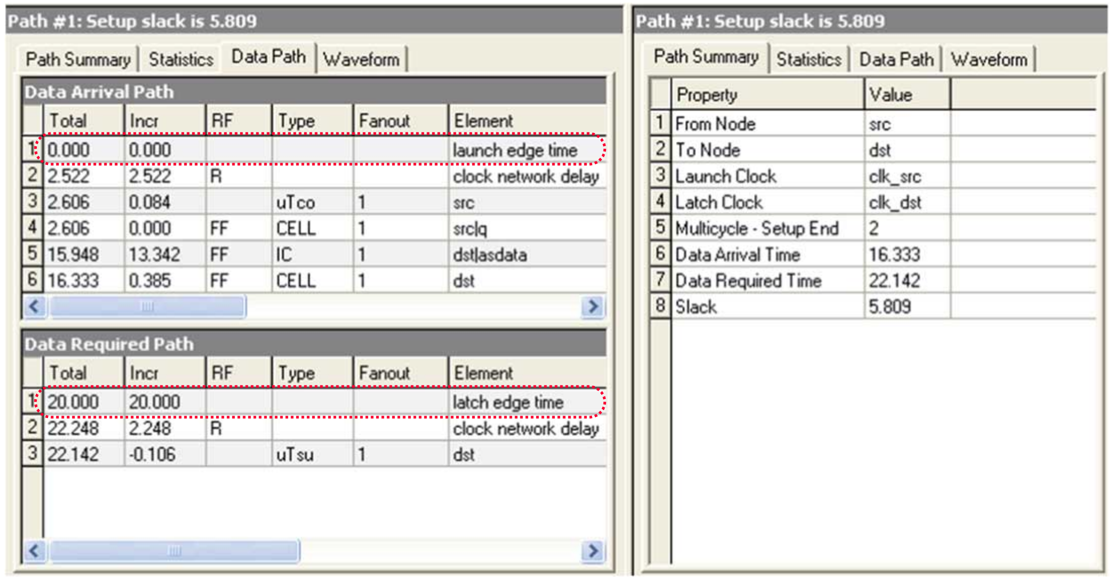
Figure 3-24: Setup Check

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

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

The setup report in the TimeQuest analyzer with the launch and latch edges highlighted.

Figure 3-25: 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

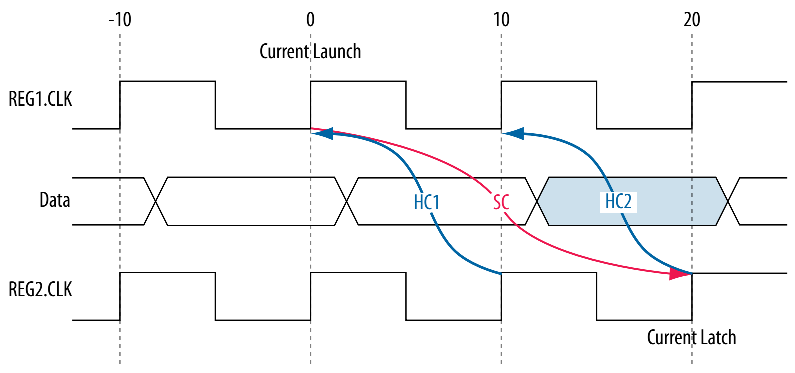
Path #1: Setup slack is 5.809

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 TimeQuest analyzer performs hold checks on every possible setup check, not only on the most restrictive setup check edges.

Figure 3-26: Hold Timing DIagram



The calculation that the TimeQuest analyzer performs to determine the hold check. Both hold checks are equivalent.

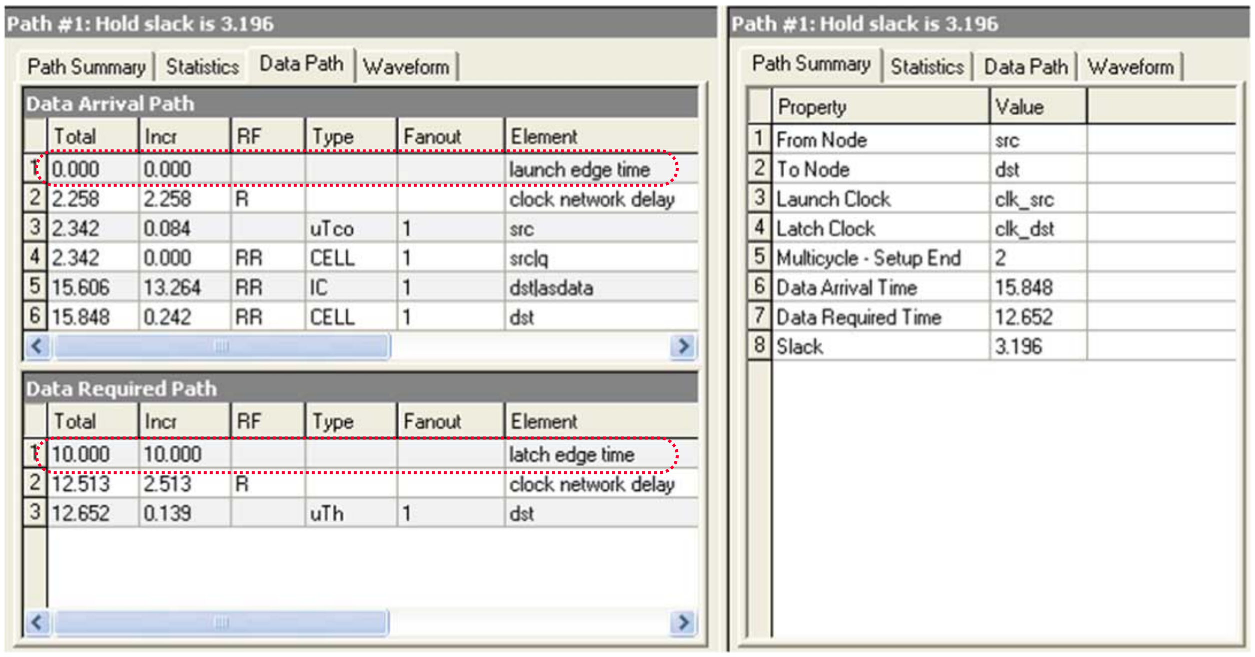
Figure 3-27:

$$
\begin{aligned}
\text{hold check 1} &= \text{current launch edge} - \text{previous latch edge} \\
&= 0\text{ ns} - 10\text{ ns} \\
&= -10\text{ ns} \\
\\
\text{hold check 2} &= \text{next launch edge} - \text{current latch edge} \\
&= 10\text{ ns} - 20\text{ ns} \\
&= -10\text{ ns}
\end{aligned}
$$

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 TimeQuest analyzer with the launch and latch edges highlighted.

Figure 3-28: 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

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.

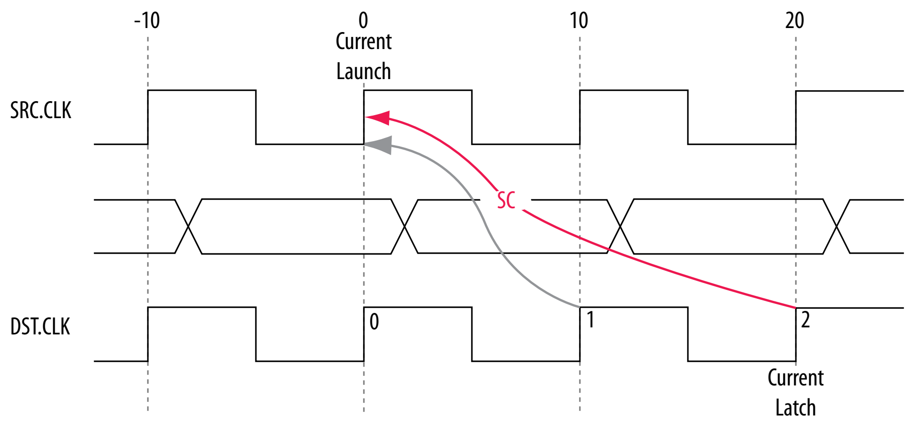
Multicycle Exceptions

```
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 TimeQuest analyzer performs.

Figure 3-29: Setup Timing Diagram



The calculation that the TimeQuest analyzer performs to determine the setup check.

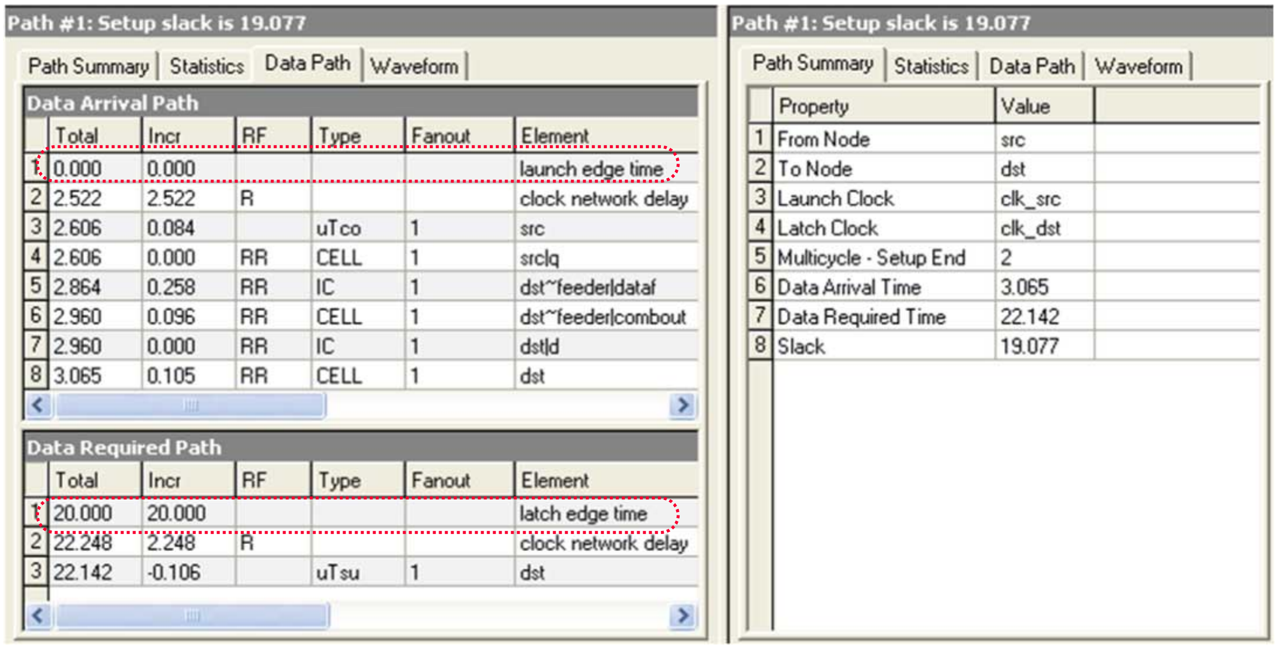
Figure 3-30: Setup Check

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

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 TimeQuest analyzer with the launch and latch edges highlighted.

Figure 3-31: 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

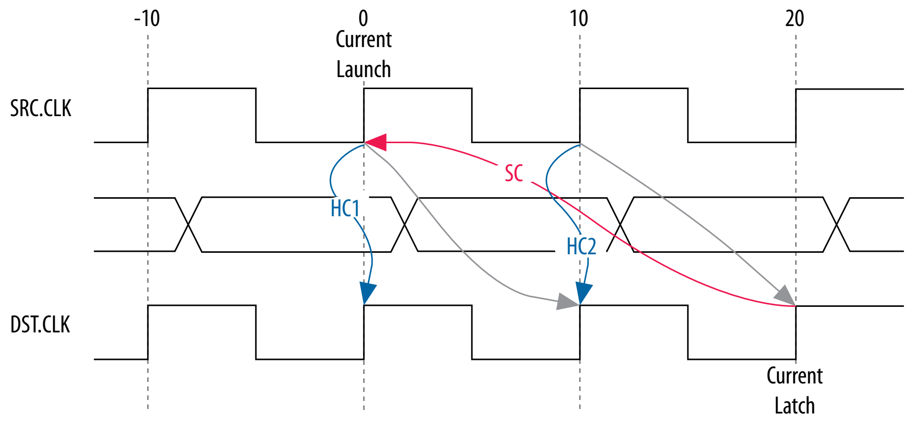
Path #1: Setup slack is 19.077

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 3-32: Hold Timing Diagram



The calculation that the TimeQuest analyzer performs to determine the hold check. Both hold checks are equivalent.

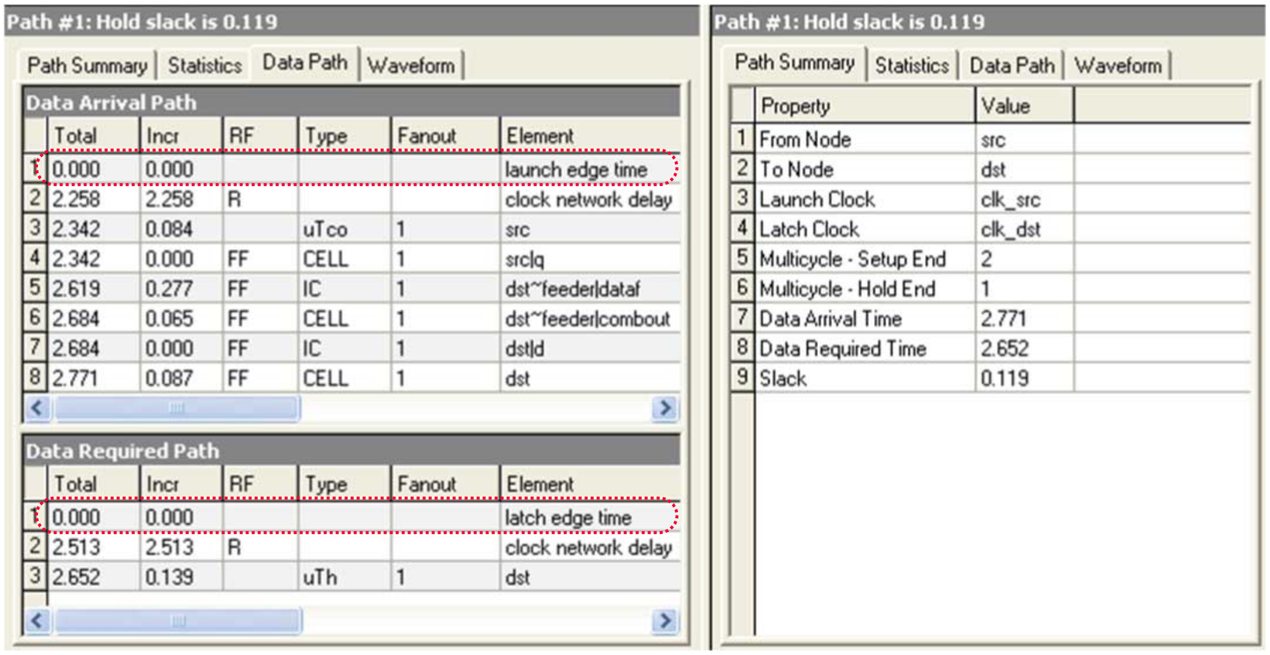
Figure 3-33: Hold Check

$$
\begin{aligned}
\text{hold check 1} &= \text{current launch edge} - \text{previous latch edge}\\
&= 0\text{ ns} - 0\text{ ns}\\
&= 0\text{ ns}\\[1em]
\text{hold check 2} &= \text{next launch edge} - \text{current latch edge}\\
&= 10\text{ ns} - 10\text{ ns}\\
&= 0\text{ ns}
\end{aligned}
$$

The most restrictive hold relationship with 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 TimeQuest analyzer with the launch and latch edges highlighted.

Figure 3-34: 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

Application of Multicycle Exceptions

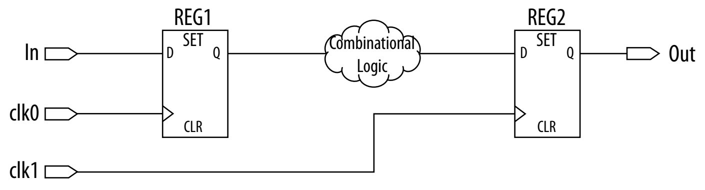
This section shows the following examples of applications of multicycle exceptions. Each example explains how the multicycle exceptions affect the default setup and hold analysis in the TimeQuest analyzer. All of the examples are between related clock domains. If your design contains related clocks, such as PLL clocks, and paths between related clock domains, you can apply multicycle constraints.

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.

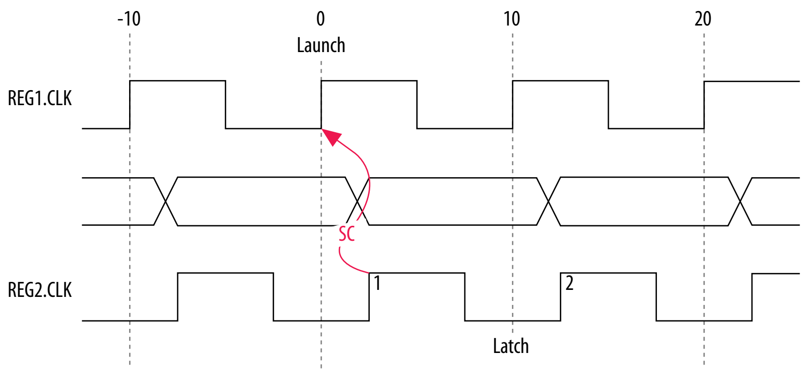
An example of a design with same frequency clocks and a destination clock offset.

Figure 3-35: Same Frequency Clocks with Destination Clock Offset



The timing diagram for default setup check analysis that the TimeQuest analyzer performs.

Figure 3-36: Setup Timing Diagram



The calculation that the TimeQuest analyzer performs to determine the setup check.

Figure 3-37: 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.

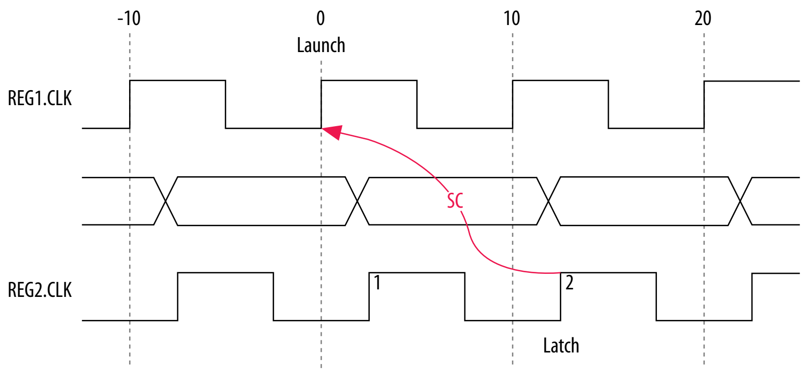
Multicycle Exceptions

```
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.

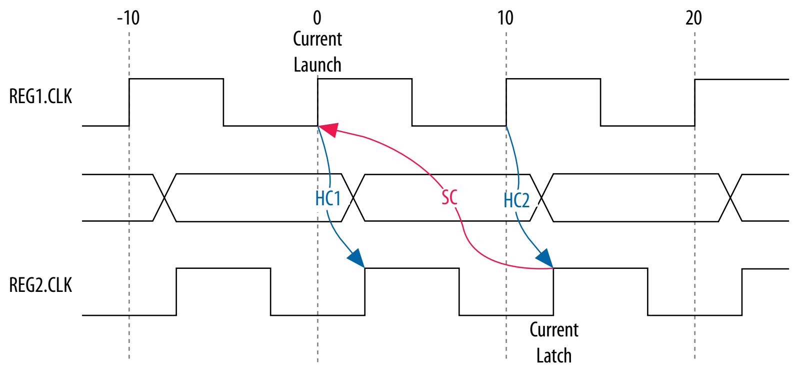
.

Figure 3-38: Preferred Setup Relationship



The timing diagram for default hold check analysis that the TimeQuest analyzer performs with an end multicycle setup value of two.

Figure 3-39: Default Hold Check



The calculation that the TimeQuest analyzer performs to determine the hold check.

Figure 3-40: Hold Check

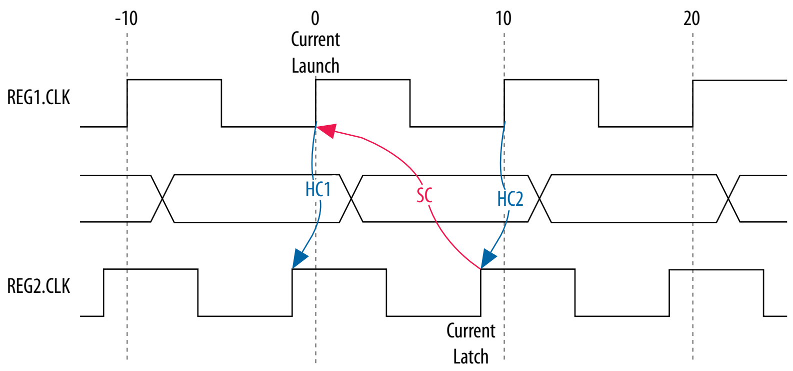
$$\begin{aligned}\text{hold check 1} &= \text{current launch edge} - \text{previous latch edge}\\ &= 0\text{ ns} - 2\text{ ns}\\ &= -2\text{ ns}\end{aligned}$$

$$\begin{aligned}\text{hold check 2} &= \text{next launch edge} - \text{current latch edge}\\ &= 10\text{ ns} - 12\text{ ns}\\ &= -2\text{ ns}\end{aligned}$$

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 3-41: Negative Phase Shift

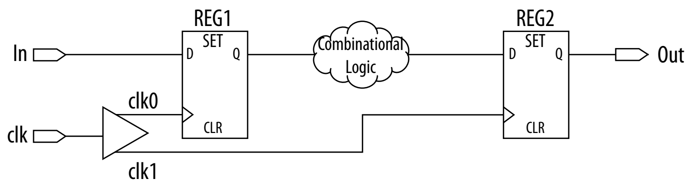


Destination Clock Frequency is a Multiple of the Source Clock Frequency

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.

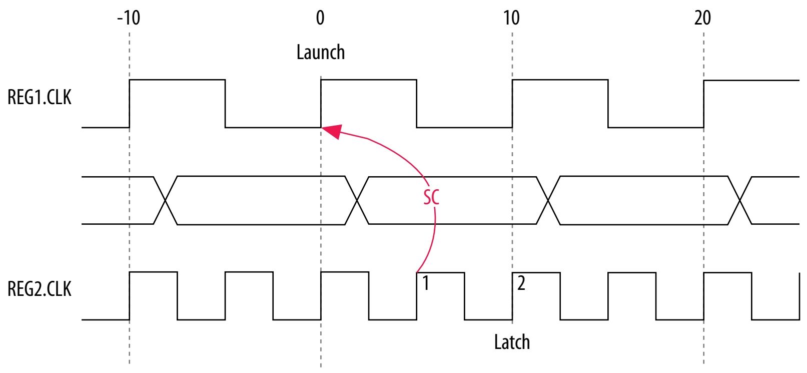
An example of a design where the destination clock frequency is a multiple of the source clock frequency.

Figure 3-42: Destination Clock is Multiple of Source Clock



The timing diagram for default setup check analysis that the TimeQuest analyzer performs.

Figure 3-43: Setup Timing Diagram



The calculation that the TimeQuest analyzer performs to determine the setup check.

Figure 3-44: Setup Check

$$\begin{aligned}\text{setup check} &= \text{current latch edge} - \text{closest previous launch edge}\\ &= 5\text{ ns} - 0\text{ ns}\\ &= 5\text{ ns}\end{aligned}$$

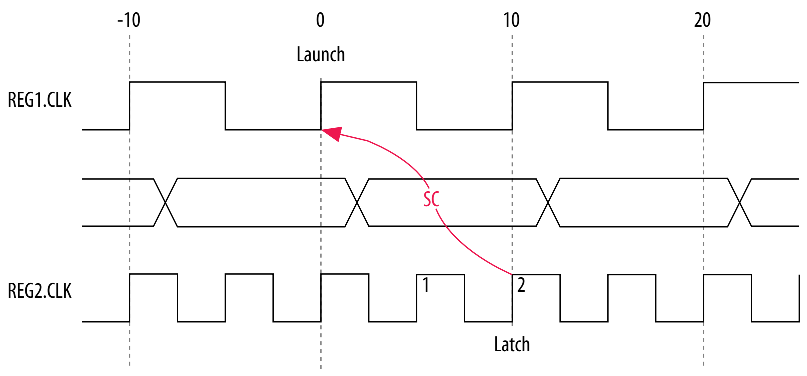
The setup relationship demonstrates that the data does not need to be captured at edge one, but can be captured at edge two; therefore, you can relax the setup requirement. To correct the default analysis, you must shift the latch edge by one clock period with an end multicycle setup exception of two. 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
```

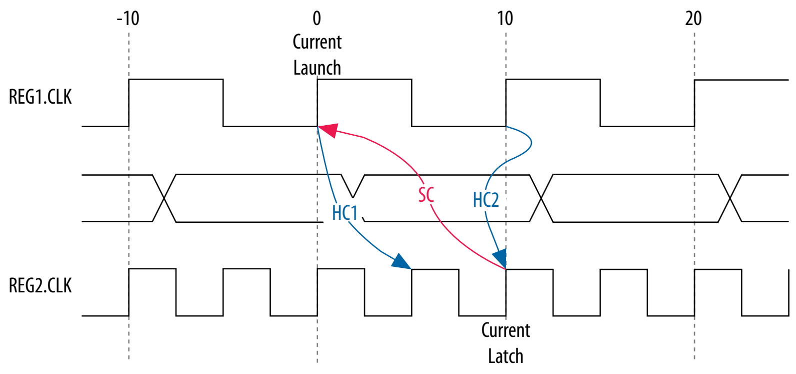
The timing diagram for the preferred setup relationship for this example.

Figure 3-45: Preferred Setup Analysis



The timing diagram for default hold check analysis performed by the TimeQuest analyzer with an end multicycle setup value of two.

Figure 3-46: Default Hold Check



The calculation that the TimeQuest analyzer performs to determine the hold check.

Figure 3-47: Hold Check

$$\begin{aligned}\text{hold check 1} &= \text{current launch edge} - \text{previous latch edge}\\ &= 0\text{ ns} - 5\text{ ns}\\ &= -5\text{ ns}\end{aligned}$$

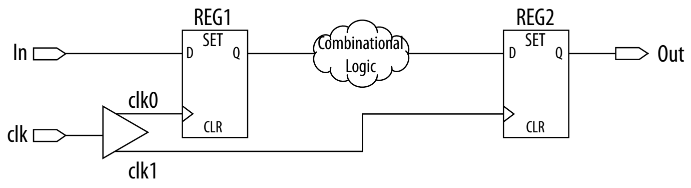
$$\begin{aligned}\text{hold check 2} &= \text{next launch edge} - \text{current latch edge}\\ &= 10\text{ ns} - 10\text{ ns}\\ &= 0\text{ ns}\end{aligned}$$

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.

Destination Clock Frequency is a Multiple of the Source Clock Frequency with an Offset

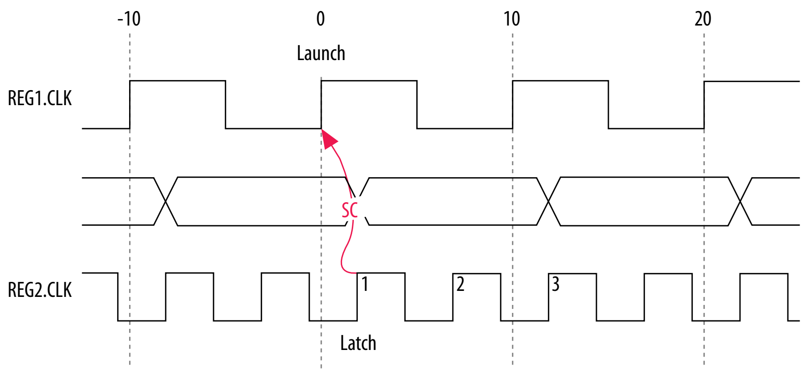
This example is a combination of the previous two examples. The destination clock frequency is an integer multiple of the source clock frequency and the destination clock has a positive phase shift. The destination clock frequency is 5 ns and the source clock frequency is 10 ns. The destination clock 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 3-48: Destination Clock is Multiple of Source Clock with Offset



The timing diagram for the default setup check analysis the TimeQuest analyzer performs.

Figure 3-49: Setup Timing Diagram



The calculation that the TimeQuest analyzer performs to determine the setup check.

Figure 3-50: 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.

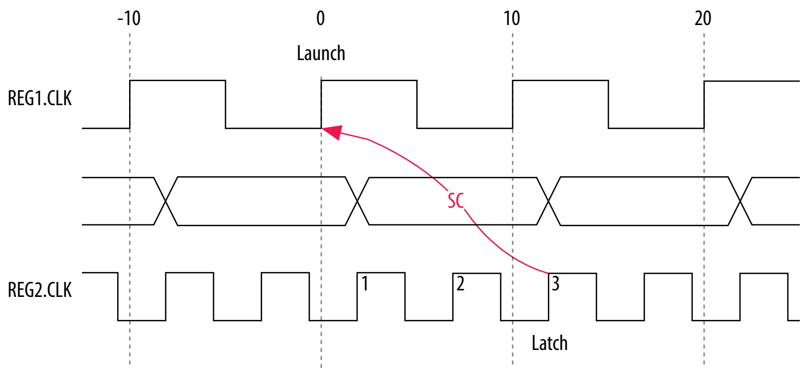
The multicycle exception code you can use to correct the default analysis in this example.

Multicycle Exceptions

```
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] \
      -setup -end 3
```

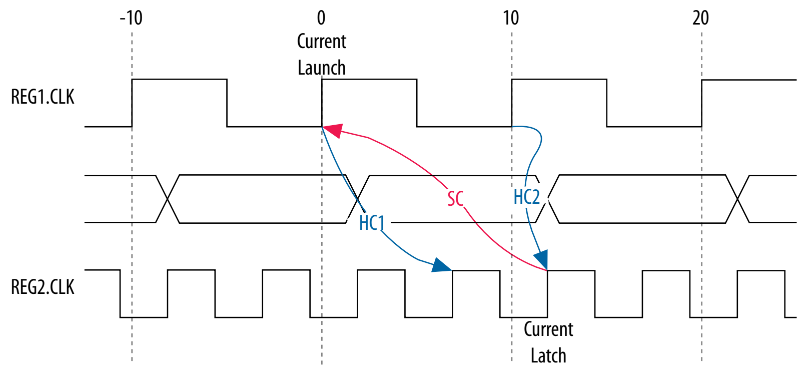
The timing diagram for the preferred setup relationship for this example.

Figure 3-51: Preferred Setup Analysis



The timing diagram for default hold check analysis the TimeQuest analyzer performs with an end multicycle setup value of three.

Figure 3-52: Default Hold Check



The calculation that the TimeQuest analyzer performs to determine the hold check.

Figure 3-53: Hold Check

$$
\begin{aligned}
\text{hold check 1} &= \text{current launch edge} - \text{previous latch edge} \\
&= 0\text{ ns} - 5\text{ ns} \\
&= -5\text{ ns} \\[1em]
\text{hold check 2} &= \text{next launch edge} - \text{current latch edge} \\
&= 10\text{ ns} - 10\text{ ns} \\
&= 0\text{ ns}
\end{aligned}
$$

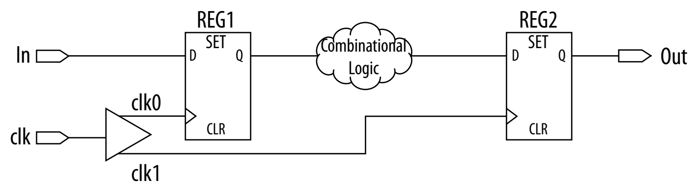
In this example, hold check one is too restrictive. The data is launched by the edge at 0 ns and should check against the data captured by the previous latch edge at 2 ns, which does not occur in hold check one. To correct the default analysis, you must use an end multicycle hold exception of one.

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.

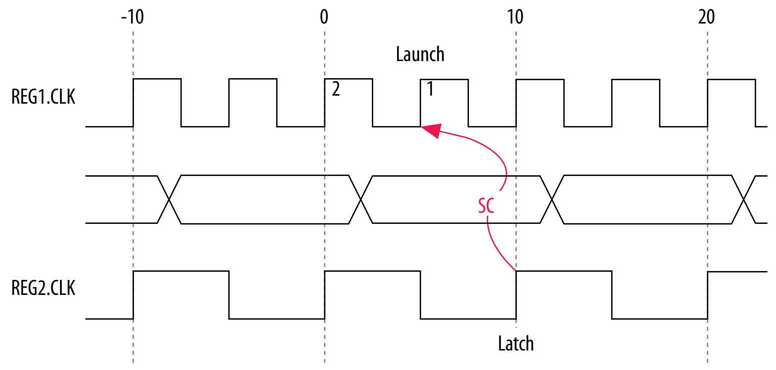
An example of a design where the source clock frequency is a multiple of the destination clock frequency.

Figure 3-54: Source Clock Frequency is Multiple of Destination Clock Frequency



The timing diagram for default setup check analysis performed by the TimeQuest analyzer.

Figure 3-55: Default Setup Check Analysis



The calculation that the TimeQuest analyzer performs to determine the setup check.

Figure 3-56: 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.

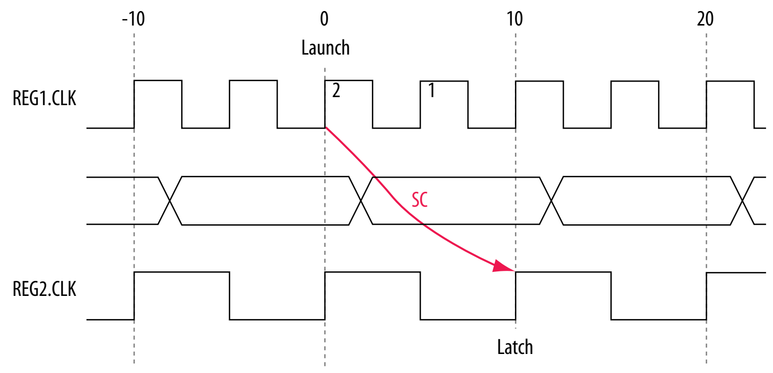
The multicycle exception code you can use to correct the default analysis in this example.

Multicycle Exceptions

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

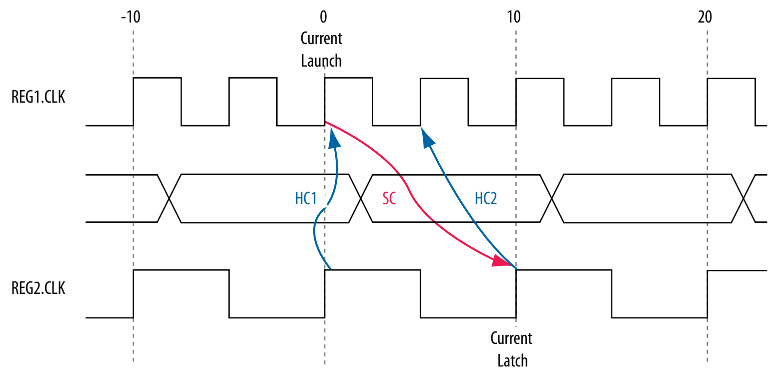
The timing diagram for the preferred setup relationship for this example.

Figure 3-57: Preferred Setup Check Analysis



The timing diagram for default hold check analysis the TimeQuest analyzer performs for a start multicycle setup value of two.

Figure 3-58: Default Hold Check



The calculation that the TimeQuest analyzer performs to determine the hold check.

Figure 3-59: Hold Check

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

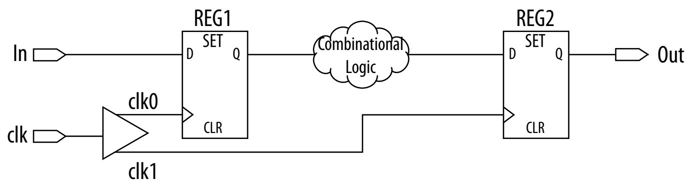
$$\begin{aligned}\text{hold check 2} &= \text{next launch edge} - \text{current latch edge}\\ &= 5\text{ ns} - 10\text{ ns}\\ &= -5\text{ ns}\end{aligned}$$

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.

Source Clock Frequency is a Multiple of the Destination Clock Frequency with an Offset

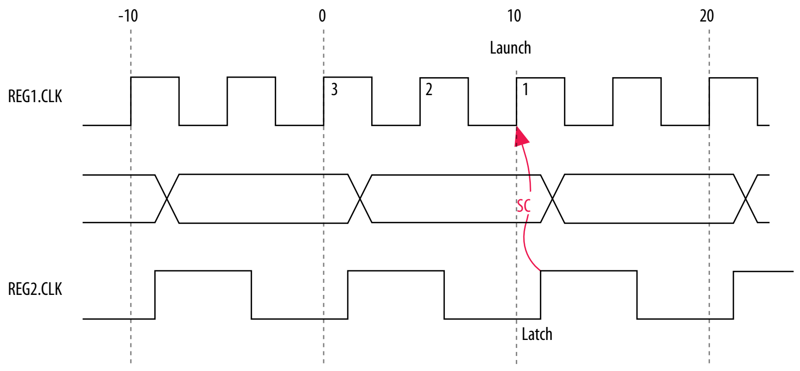
In this example, the source clock frequency is an integer multiple of the destination clock frequency and the destination clock has a positive phase offset. The source clock frequency is 5 ns and destination clock frequency is 10 ns. The destination clock also has a positive offset of 2 ns with respect 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 3-60: Source Clock Frequency is Multiple of Destination Clock Frequency with Offset



Timing diagram for default setup check analysis the TimeQuest analyzer performs.

Figure 3-61: Setup Timing Diagram



The calculation that the TimeQuest analyzer performs to determine the setup check.

Figure 3-62: 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.

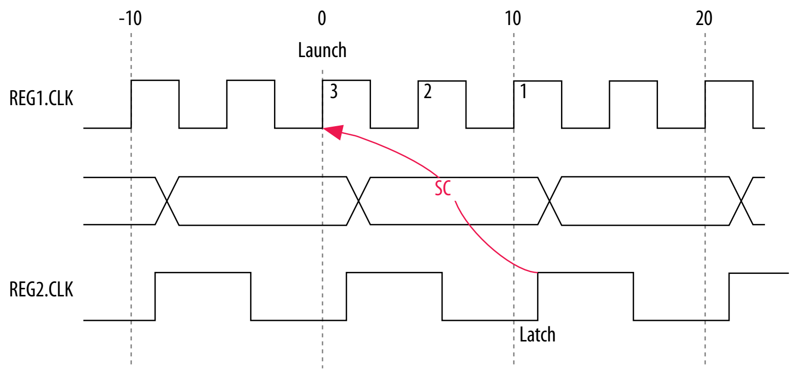
The multicycle exception used to correct the default analysis in this example.

Multicycle Exceptions

```
set_multicycle_path -from [get_clocks clk_src] -to [get_clocks clk_dst] \
     -setup -start 3
```

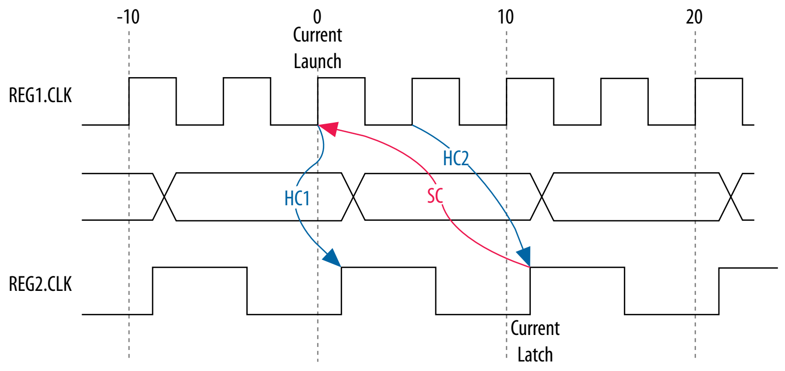
The timing diagram for the preferred setup relationship for this example.

Figure 3-63: Preferred Setup Check Analysis



The timing diagram for default hold check analysis the TimeQuest analyzer performs for a start multicycle setup value of three.

Figure 3-64: Default Hold Check Analysis



The calculation that the TimeQuest analyzer performs to determine the hold check.

Figure 3-65: Hold Check

hold check 1 = current launch edge – previous latch edge
= 0 ns – 2 ns
= –2 ns

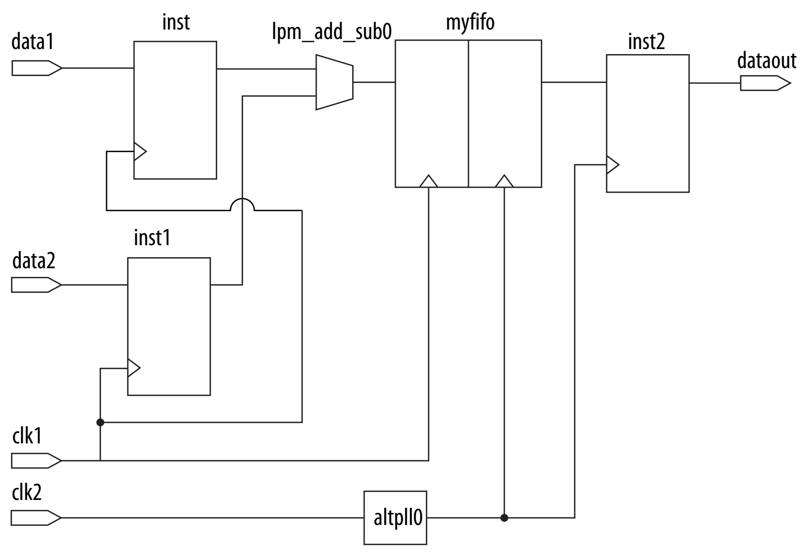
hold check 2 = next launch edge – current latch edge
= 5 ns – 12 ns
= –7 ns

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.

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 3-66: TimeQuest Constraint Example



The following SDC file contains basic constraints for the example circuit.

Example 3-5: 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 clockone_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 Information

Asynchronous Clock Groups on page 3-21
For more information about asynchronous clock groups.

Running the TimeQuest Analyzer

When you compile a design, the TimeQuest timing analyzer automatically performs multi-corner signoff timing analysis after the Fitter has finished.

- To open the TimeQuest analyzer GUI directly from the Quartus Prime software GUI, click **TimeQuest Timing Analyzer** on the Tools menu.
- To peform or repeat multi-corner timing analysis from the Quartus Prime GUI, click **Processing** > **Start** > **Start TimeQuest Timing Analyzer**.
- To perform multi-corner timing analysis from a system command prompt, type `quartus_sta` `<`*`options`*`><`*`project_name`*`>`.
- To run the TimeQuest analyzer as a stand-alone GUI application, type the following command at the command prompt:`quartus_staw`.
- To run the TimeQuest 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 3-9: 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 TimeQuest 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 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.

Command-Line Option	Description
`--64bit`	Enables 64-bit version of the executable.
`--mode=implement|finalize`	Regulates whether TimeQuest performs a reduced-corner analysis for intermediate operations (`implement`),or a four-corner analysis for final Fitter optimization and placement (`finalize`).

Related Information

- **Constraining and Analyzing with Tcl Commands** on page 3-65
  For more information about using Tcl commands to constrain and analyze your design
- **Recommended Flow for First Time Users** on page 3-2
  For more information about steps to perform before opening the TimeQuest analyzer.

Quartus Prime Settings

Within the Quartus Prime software, there are a number of quick steps for setting up your design with TimeQuest. You can modify the appropriate settings in **Assignments** > **Settings**.

In the **Settings** dialog box, select **TimeQuest Timing Analyzer** in the **Category** list.

The **TimeQuest 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 Quartus Prime software provides an SDC template you can use to create your own.

The following TimeQuest options should be on by default:

- Enable multicorner timing analysis—Directs the TimeQuest 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.

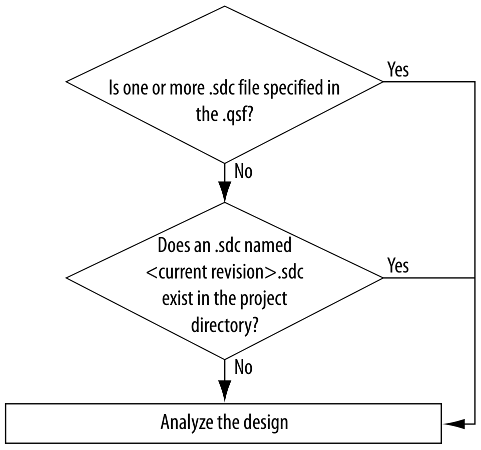
SDC File Precedence

The Fitter and the TimeQuest analyzer process SDC files in the order you specify in the 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 **TimeQuest 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 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 Quartus Prime software looks for an SDC named ***<current revision>*.sdc** in the project directory. When you use IP from Altera, and some third-parties, the SDC files are often included in a project through an intermediate file called a 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 Quartus Prime IP File (**.qip**) included in the **.qsf**.

Figure 3-67: .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 TimeQuest 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.

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.

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 they will be applied appropriately during compilation and timing analysis.

Use the following steps when you iteratvely modify constraints:

1. Open the TimeQuest 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 TimeQuest Timing Analyzer—It is most common to use this interactive approach in the TimeQuest 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 on 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 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 TimeQuest 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 Quartus Prime software can do with those constraints.

Related Information

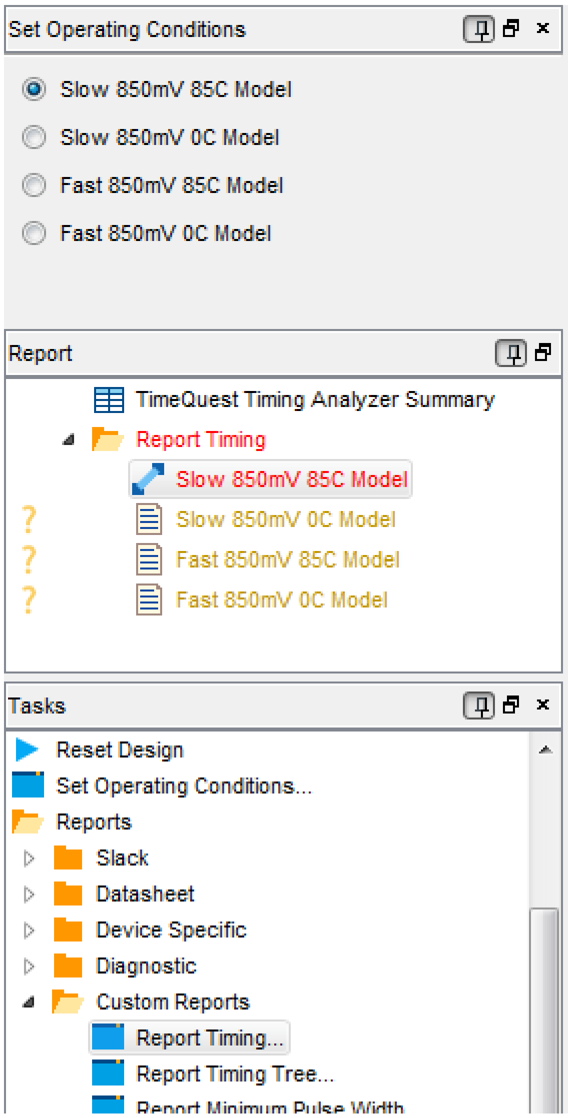
Relaxing Setup with set_multicyle_path on page 3-31

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 TimeQuest 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 TimeQuest 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.


Set Operating Conditions

- [✓ Slow 850mV 85C Model]
- Slow 850mV 0C Model
- Fast 850mV 85C Model
- Fast 850mV 0C Model

Report

- TimeQuest Timing Analyzer Summary
- Report Timing
  - Slow 850mV 85C Model
  - ? Slow 850mV 0C Model
  - ? Fast 850mV 85C Model
  - ? Fast 850mV 0C Model

Tasks

- Reset Design
- Set Operating Conditions...
- Reports
  - Slack
  - Datasheet
  - Device Specific
  - Diagnostic
  - Custom Reports
    - Report Timing...
    - Report Timing Tree...
    - Report Minimum Pulse Width

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.

Report Timing (Dialog Box)

Once you are comfortable with the **Report All Summaries** command, the next tool in the TimeQuest analyzer toolbox is **Report Timing...**.

The TimeQuest 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. Below is a design with the **Summary (Setup)** report highlighted:

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 TimeQuest analyzer UI. This is often done as users become more comfortable with TimeQuest and find themselves analyzing the same problematic parts of their design over and over, but is not required. Many complex designs successfully use TimeQuest as a diving tool, i.e. just starting with summaries and diving down into the failing paths after each compile.

Analyzing Results with Report Timing

Report Timing is one of the most useful analysis tools in TimeQuest. Many designs require nothing but this command. In the TimeQuest 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 TimeQuest.

You can review all of the options for **Report Timing** by typing `report_timing -long_help` in the TimeQuest 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 TimeQuest 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**. These will be explained in more detail later, as understanding them is the underpinning of timing analysis.

Output

The **Detail** level, is an option often glanced over that should be understood. It has four options, but I will only discuss three.

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 TimeQuest'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 TimeQuest 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 TimeQuest. 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.

Correlating Constraints to the Timing Report

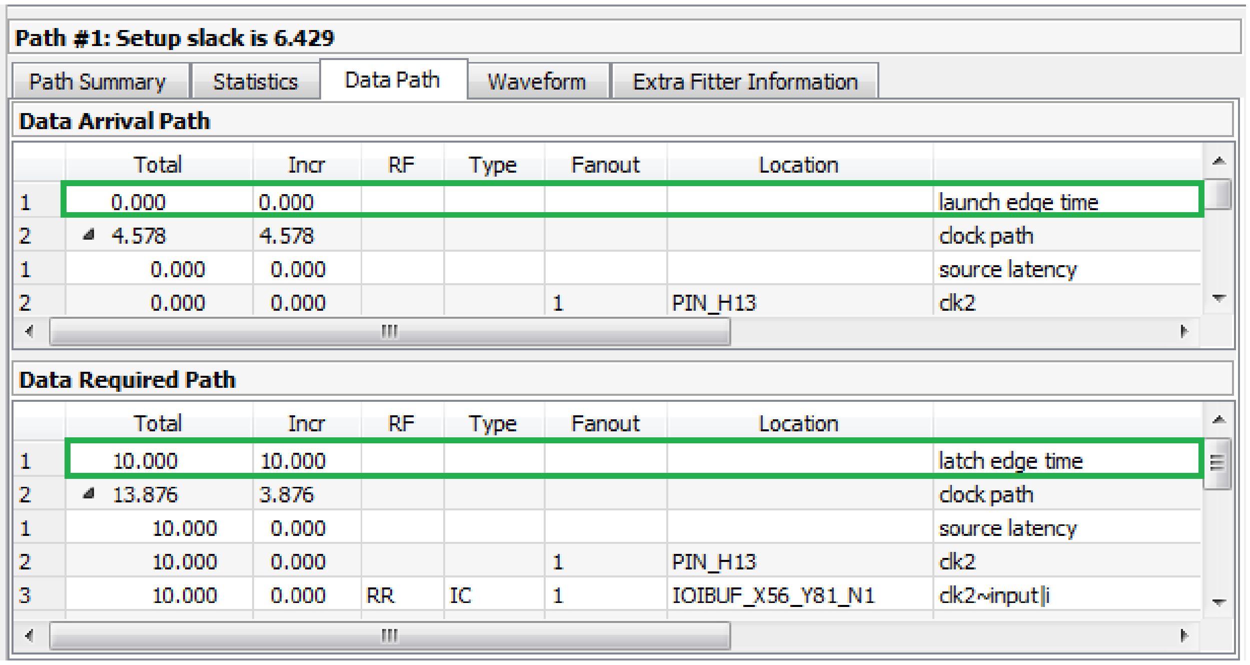
A critical part of timing analysis is how timing constraints appear in the **Report Timing** analysis. Most constraints only affect the launch and latch edges. Specifically, `create_clock` and `create_generated_clock` create clocks with default relationships. The command `set_multicycle_path` will modify those default relationships, while `set_max_delay` and `set_min_delay` are low-level overrides that explicitly tell TimeQuest what the launch and latch edges should be.

The folowing 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 3-68: Setup Relationship 10ns, Hold Relationship 0ns



Path #1: Setup slack is 6.429

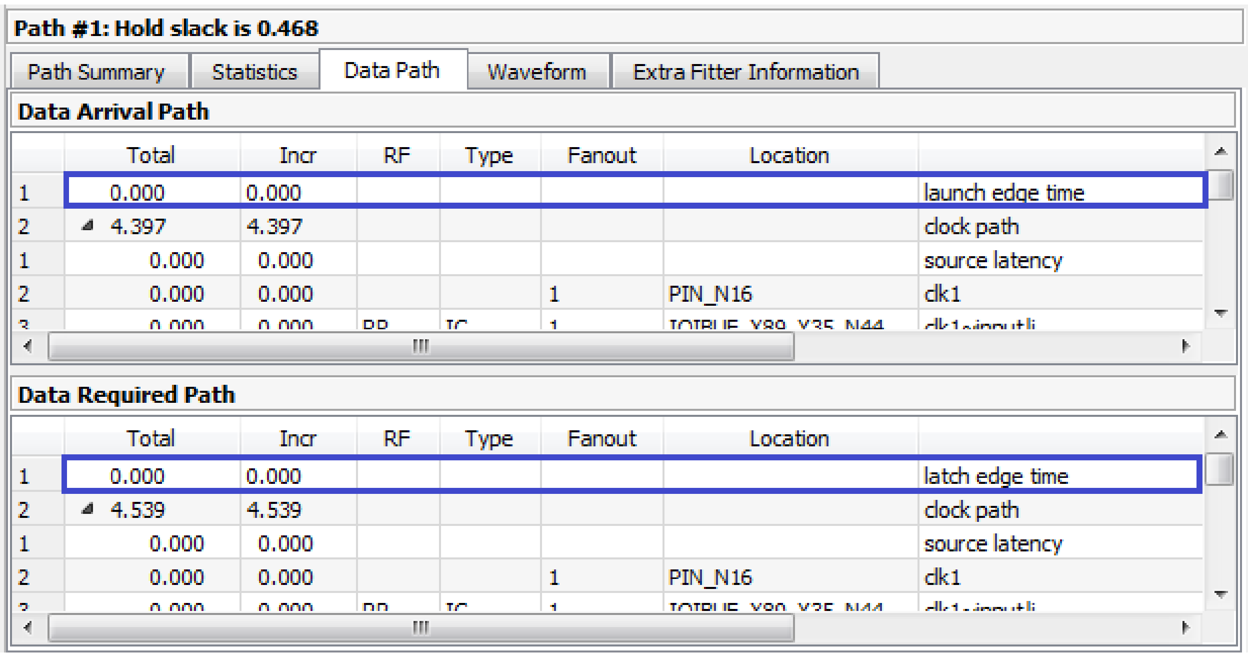
Path Summary | Statistics | Data Path | Waveform | Extra Fitter Information

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

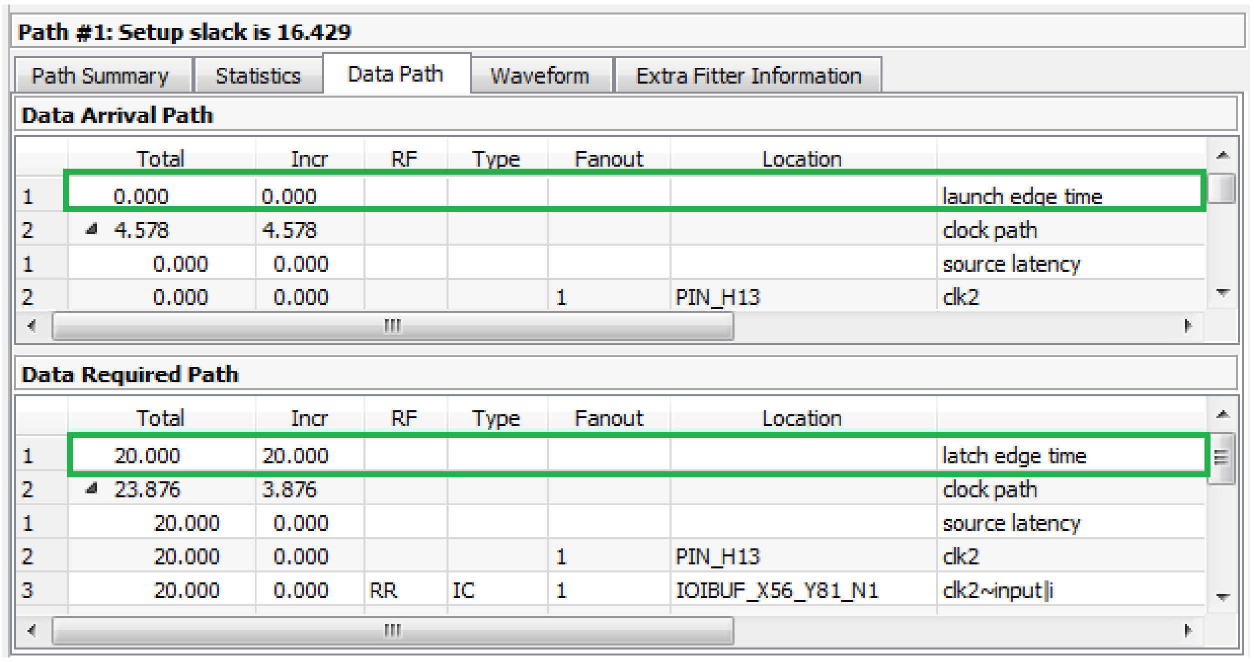
Data Required Path

	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 3-69: Setup Relationship 20ns



Path #1: Setup slack is 16.429

Path Summary | Statistics | Data Path | Waveform | Extra Fitter Information

Data Arrival Path

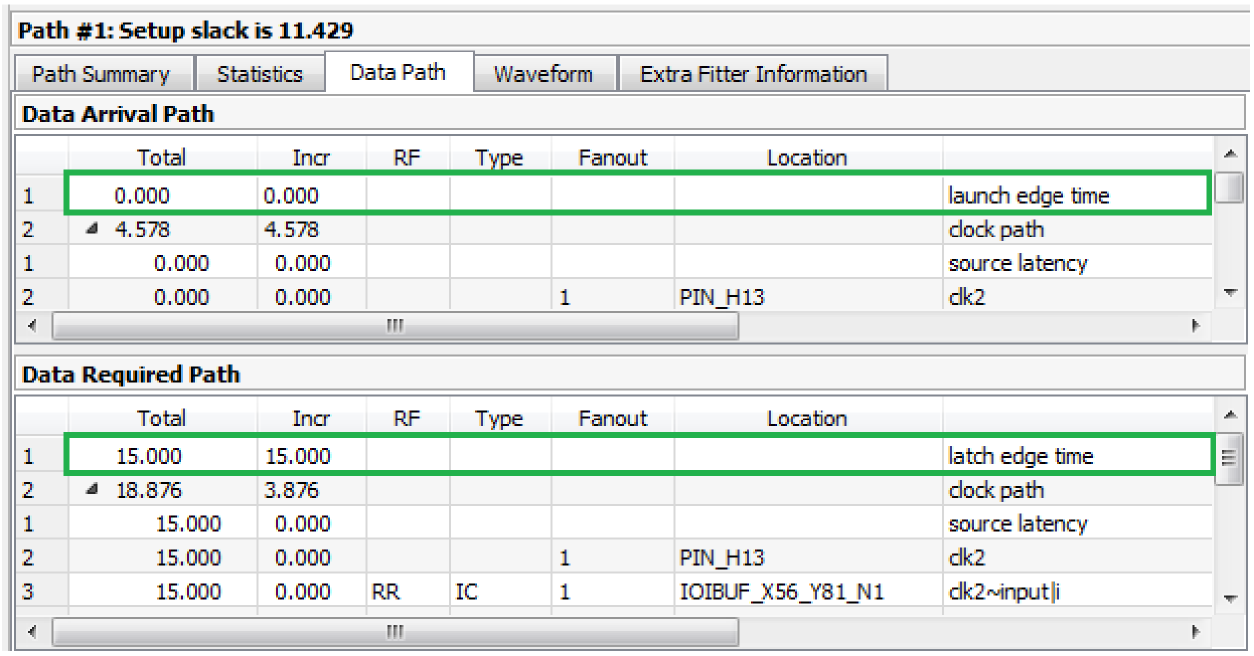
	Total	Incr	RF	Type	Fanout	Location	
1	0.000	0.000					launch edge time
2	◢ 4.578	4.578					clock path
1	0.000	0.000					source latency
2	0.000	0.000			1	PIN_H13	clk2

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 3-70: Using set_max_delay and set_min_delay



Path #1: Setup slack is 11.429

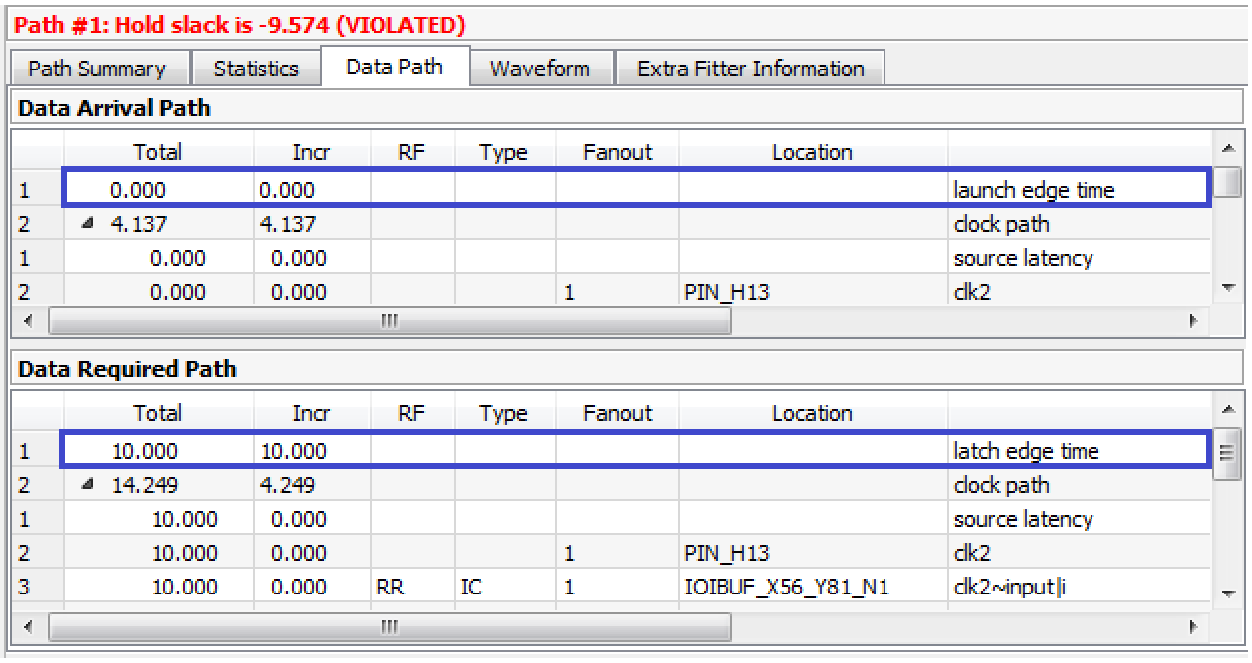
Path Summary | Statistics | Data Path | Waveform | Extra Fitter Information

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	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 will subtract 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 Information

Relaxing Setup with set_multicyle_path on page 3-31

Constraining and Analyzing with Tcl Commands

You can use Tcl commands from the 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 TimeQuest analyzer GUI. SDC commands are Tcl commands for constraining a design. SDC extension commands provide additional constraint methods and are specific to the TimeQuest analyzer. Additional TimeQuest analyzer commands are available for controlling timing analysis and reporting. These commands are contained in the following Tcl packages available in the Quartus Prime software:

- `::quartus::sta`
- `::quartus::sdc`
- `::quartus::sdc_ext`

Related Information

- **::quartus::sta**
  For more information about TimeQuest analyzer Tcl commands and a complete list of commands, refer to Quartus Prime Help.
- **::quartus::sdc**
  For more information about standard SDC commands and a complete list of commands, refer to Quartus Prime Help.
- **::quartus::sdc_ext**
  For more information about Altera extensions of SDC commands and a complete list of commands, refer to Quartus Prime Help.

Collection Commands

The TimeQuest 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 TimeQuest 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 TimeQuest analyzer.

Table 3-10: 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.
`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 TimeQuest analyzer by clicking **Name Finder** from the **View** menu.

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 TimeQuest 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 TimeQuest 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`.

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 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 TimeQuest analyzer flow and is not compatible with third-party timing analysis flows.

Related Information

- add_to_collection
- remove_from_collection
  For more information about theadd_to_collection and remove_from_collection commands, refer to Quartus Prime Help.

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]]

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 3-11: 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`

[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.

Search String	Search Result
`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`.

Identifying the Quartus Prime Software Executable from the SDC File

To identify which 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.

Identifying the 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 TimeQuest analyzer.

Locating Timing Paths in Other Tools

You can locate paths and elements from the TimeQuest analyzer to other tools in the Quartus Prime software.

[4] When you use `-compatibility_mode`, pipe characters (|) are not treated as special characters when used with wildcards.

Use the **Locate** or `Locate Path` command in the TimeQuest analyzer GUI or the `locate` command in the Tcl console in the TimeQuest analyzer GUI. Right-click on most paths or node names in the TimeQuest 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 TimeQuest analyzer to the **Technology Map Veiwer** and locate all ports matching data* in the **Chip Planner**.

Example 3-6: Locating from the TimeQuest 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 Information

- **Viewing Timing Analysis Results**
  For more information about locating paths from the TimeQuest analyzer, refer to Quartus Prime Help.
- **locate**
  For more information on this command, refer to Quartus Prime Help.

Generating Timing Reports

The TimeQuest analyzer provides real-time static timing analysis result reports. The TimeQuest analyzer does not automatically generate most reports; you must create each report individually in the TimeQuest 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 TimeQuest analyzer and the equivalent reports shown in the TimeQuest analyzer GUI.

Table 3-12: TimeQuest 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 Quartus Prime software generates timing reports on different timing areas in the design. You can configure various options for the TimeQuest 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.

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

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 Information

- **::quartus::sta**
  For more information on this command, refer to Quartus Prime Help.
- **TimeQuest Timing Analyzer Page**
  For more information about the options you can set to customize TimeQuest analyzer reports.
- **Area and Timing Optimization**
  For more information about timing closure recommendations.

Document Revision History

Table 3-13: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	• Changed instances of *Quartus II* to *Quartus Prime*. • Added a description of running three- and four-corner analysis with `--mode=implement|finalize`. • Added description for new `set_operating_conditions` UI.

Date	Version	Changes
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 TimeQuest 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 Quartus Prime Templates with the Quartus Prime Text Editor" section on creating an SDC constraints file with the **Insert Template** dialog box. • Added "Identifying the 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 Quartus Prime TimeQuest Timing Analyzer chapter. • Changed to new document template.
May 2011	11.0.0	• Updated to improve flow. Minor editorial updates.

Date	Version	Changes
December 2010	10.1.0	• Changed to new document template. • Revised and reorganized entire chapter. • Linked to Quartus Prime Help.
July 2010	10.0.0	Updated to link to content on SDC commands and the TimeQuest analyzer GUI in Quartus Prime Help.
November 2009	9.1.0	Updated for the 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 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 Information

Quartus Handbook Archive

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.

PowerPlay Power Analysis 4

2015.11.02

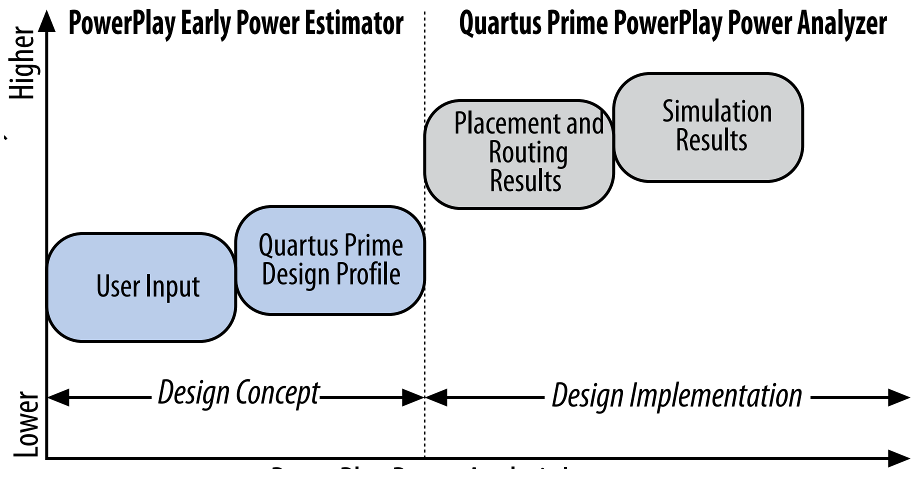
QPP5V3 Subscribe Send Feedback

The PowerPlay 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 PowerPlay Power Analysis tools ability to estimate power consumption from early design concept through design implementation.

Figure 4-1: PowerPlay Power Analysis From Design Concept Through Design Implementation


PowerPlay Early Power Estimator
Quartus Prime PowerPlay Power Analyzer
Higher
Lower
Placement and Routing Results
Simulation Results
User Input
Quartus Prime Design Profile
Design Concept
Design Implementation

For the majority of the designs, the PowerPlay Power Analyzer and the PowerPlay EPE spreadsheet have the following accuracy after the power models are final:

- PowerPlay Power Analyzer—±20% from silicon, assuming that the PowerPlay Power Analyzer uses the Value Change Dump File (**.vcd**) generated toggle rates.
- PowerPlay EPE spreadsheet— ±20% from the PowerPlay 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 PowerPlay Power Analyzer with a **.vcd** file generated from a gate level simulation representative of the system operation.

Related Information

- PowerPlay Early Power Estimators (EPE) and Power Analyzer

Types of Power Analyses

Understanding the uses of power analysis and the factors affecting power consumption helps you to use the PowerPlay 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 PowerPlay EPE at the early stages of your design cycle. Use the PowerPlay 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.

Related Information

- PowerPlay Early Power Estimator (EPE) User Guide

Differences between the PowerPlay EPE and the *Quartus Prime* PowerPlay Power Analyzer

The following table lists the differences between the PowerPlay EPE and the Quartus Prime PowerPlay Power Analyzer.

Table 4-1: Comparison of the PowerPlay EPE and Quartus Prime PowerPlay Power Analyzer

Characteristic	PowerPlay EPE	Quartus Prime PowerPlay Power Analyzer
Phase in the design cycle	Any time, but it is recommended to use Quartus Prime PowerPlay Power Analyzer for post-fit power analysis.	Post-fit
Tool requirements	Spreadsheet program	The 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 • Register transfer level (RTL) simulation results (optional) • Post-fit simulation results (optional) • Signal activities per node or entity (optional)
Data outputs *(1)*	• 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 PowerPlay Power Analyzer is only an estimation of power. Altera 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.

[5] PowerPlay EPE and PowerPlay Power Analyzer outputs vary by device family. For more information, refer to the device-specific PowerPlay Early Power Estimators (EPE) and Power Analyzer Page and PowerPlay Power Analyzer Reports in the Quartus Prime Help.

Related Information

- **PowerPlay Early Power Estimators (EPE) and Power Analyzer Page**
  For more information, refer to the device-specific PowerPlay Early Power Estimators (EPE) page on the Altera website.
- **PowerPlay Power Analyzer Reports**
  For more information, refer to this page for device-specific information about the PowerPlay Early Power Estimator.

Factors Affecting Power Consumption

Understanding the following factors that affect power consumption allows you to use the PowerPlay Power Analyzer and interpret its results effectively:

- **Device Selection**
- **Environmental Conditions**
- **Device Resource Usage**
- **Signal Activities**

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.

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.

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 4-2: 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.

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 Quartus Prime software automatically disables unused branches of a clock.

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 4-3: Signal Behavior

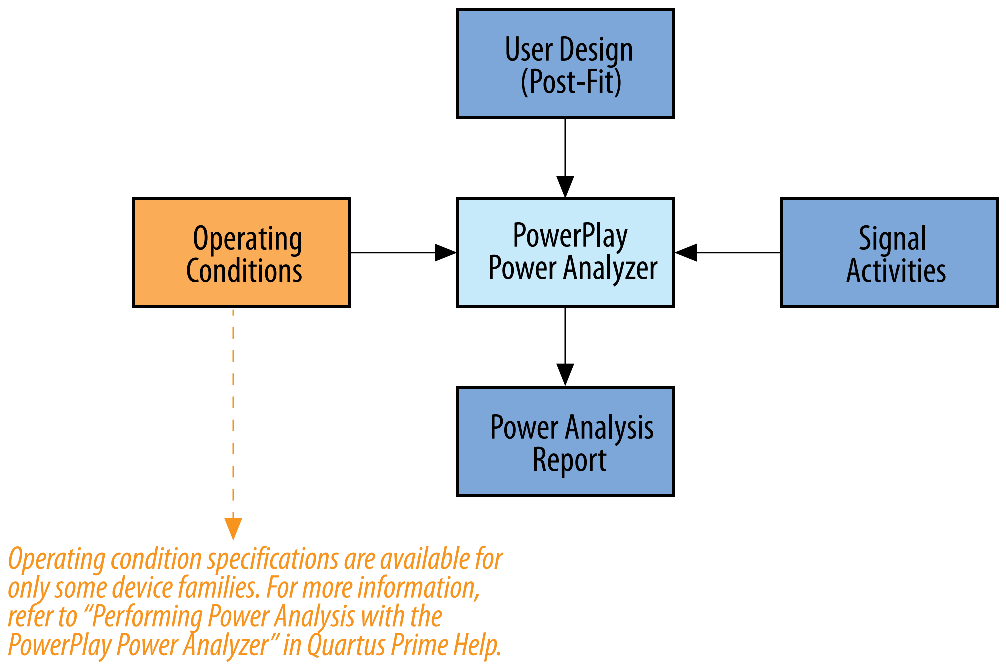
Signal Behavior	Description
Toggle rate	• The toggle rate of a signal is the average number of times that the signal changes value per unit of time. The units for toggle rate are transitions per second and a transition is a change from `1` to `0`, or `0` to `1`. • Dynamic power increases linearly with the toggle rate as you charge the board trace model more frequently for logic and routing. The 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 PowerPlay 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 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.

PowerPlay Power Analyzer Flow

The PowerPlay 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 PowerPlay Power Analyzer flow.

Figure 4-2: PowerPlay Power Analyzer High-Level Flow


User Design (Post-Fit)
Operating Conditions
PowerPlay 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 PowerPlay Power Analyzer" in Quartus Prime Help.

To obtain accurate I/O power estimates, the PowerPlay 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.

Operating Settings and Conditions

You can specify device power characteristics, operating voltage conditions, and operating temperature conditions for power analysis in the 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 Quartus Prime Fitter may override some of the supply voltages settings specified in this chapter. For example, supply voltages for several Stratix V 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 Quartus Prime PowerPlay 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 Information

- Operating Settings and Conditions Page (Settings Dialog Box)
- Voltage Page (Settings Dialog Box)
- Temperature Page (Settings Dialog Box)

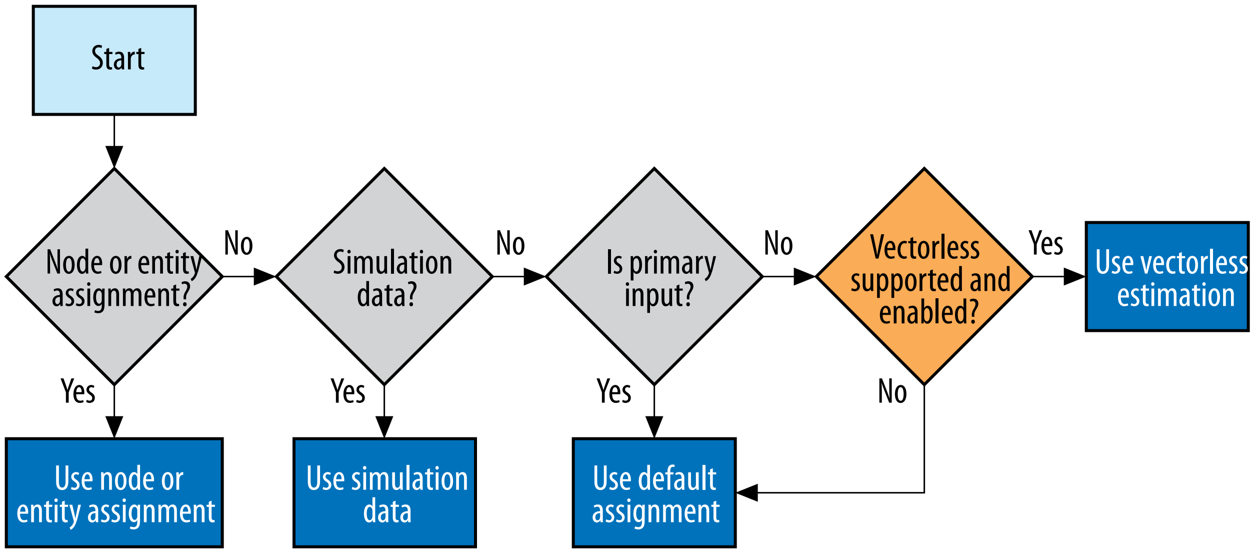
Signal Activities Data Sources

The PowerPlay 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 PowerPlay 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 4-3: Signal-Activity Data Source Priority Scheme


Start
Node or entity assignment?
No
Yes
Use node or entity assignment
Simulation data?
No
Yes
Use simulation data
Is primary input?
No
Yes
Use default assignment
Vectorless supported and enabled?
Yes
No
Use vectorless estimation

Simulation Results

The PowerPlay 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 PowerPlay Power Analyzer reads results generated by the following simulators:

- ModelSim®
- ModelSim-Altera
- 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 4-6.

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 Quartus Prime PowerPlay Power Analyzer reads this file directly to derive the toggle rate and static probability data for each signal.

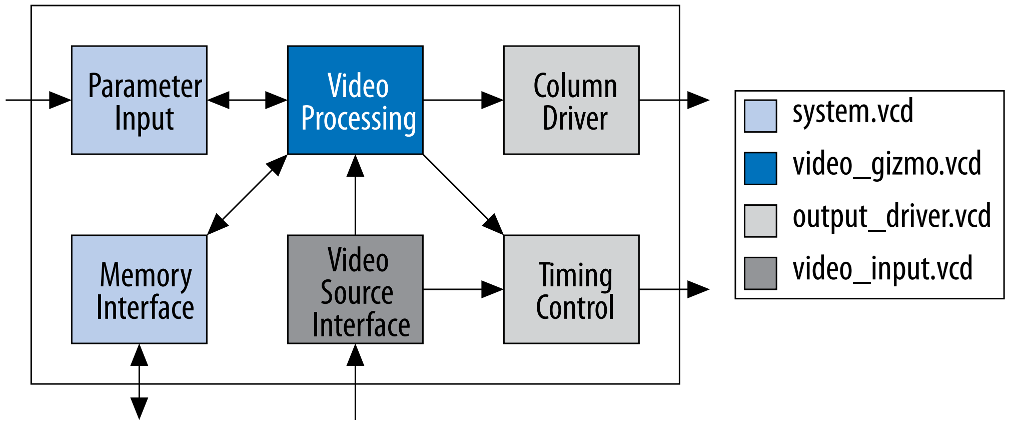
Third-party EDA simulators, other than those listed, can generate a **.vcd** that you can use with the PowerPlay Power Analyzer. For those simulators, you must manually create a simulation script to generate the appropriate **.vcd**.

Note: You can use a **.vcd** created for power analysis to optimize your design for power during fitting by utilizing the appropriate settings in the PowerPlay power optimization list, available from **Assignments** > **Settings** > **Compiler Settings** > **Advanced Settings (Fitter)**.

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 PowerPlay 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 4-4: Modular Simulation Flow



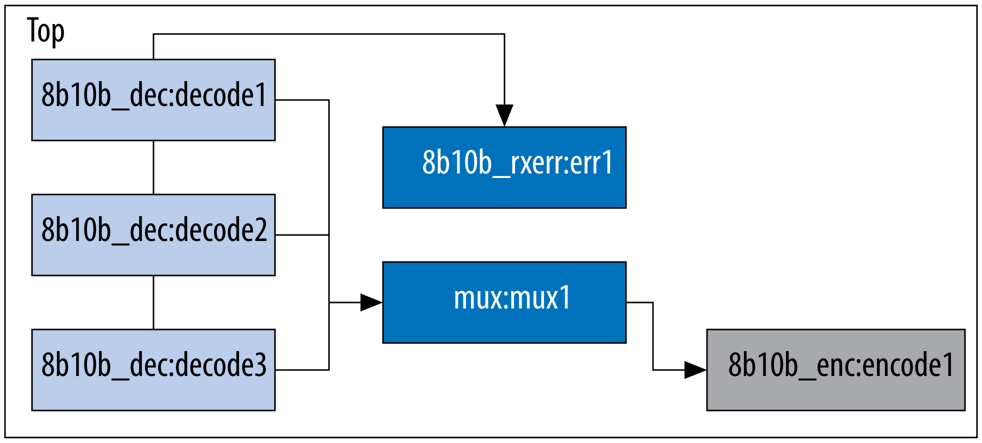
When specifying a simulation file (a **.vcd**), the software provides support to specify an associated design entity name, such that the PowerPlay Power Analyzer imports the signal activities derived from that file for the specified design entity. The PowerPlay Power Analyzer also supports the specification 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 PowerPlay Power Analyzer.

Figure 4-5: Example Hierarchical Design



Complete Design Simulation

You can simulate the entire design and generate a **.vcd** from a third-party simulator. The PowerPlay 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`.

Modular Design Simulation

You can independently simulate of the top-level design, and then import all the resulting **.vcd** files into the PowerPlay 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 4-4: 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.

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 4-5: 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 PowerPlay 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.

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 4-6: 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.

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 PowerPlay 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.

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 Quartus Prime software, on the Assignments menu, click **Settings**.
2. Under the Category list, click **PowerPlay 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 Information

- set_power_file_assignment
- Add/Edit Power Input File Dialog Box

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 Quartus Prime projects might not match their node names with the current Quartus Prime project.

For example, you may have a file named **8b10b_enc.vcd**, which the 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 Quartus Prime software might name all the combinational nodes in the **8b10b_enc.vcd** differently than in the **Top** project.

Glitch Filtering

The PowerPlay 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: Altera recommends that you use the transport simulation model when using the 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: Altera recommends that you use the glitch filtering feature to obtain the most accurate power estimates. For **.vcd** files, the PowerPlay Power Analyzer flows support two levels of glitch filtering.

Enabling First Level of Glitch Filtering

To enable the first level of glitch filtering in the 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**.

Enabling Second Level of Glitch Filtering

The second level of glitch filtering occurs while the PowerPlay 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 **PowerPlay 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 PowerPlay Power Analyzer), Altera 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 PowerPlay Power Analyzer, Altera recommends that you turn on the glitch filtering to produce the most accurate power estimates.

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 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 PowerPlay 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 PowerPlay 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.

Related Information

Constraining Designs

For more information about how to use the Assignment Editor in the Quartus Prime software, refer to this document.

Timing Assignments to Clock Nodes

For clock nodes, the PowerPlay 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.

Default Toggle Rate Assignment

You can specify a default toggle rate for primary inputs and other nodes in your design. The PowerPlay Power Analyzer uses the default toggle rate when no other method specifies the signal-activity data.

The PowerPlay 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 PowerPlay 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 PowerPlay Power Analyzer cannot determine the clock domain for a node because either the PowerPlay Power Analyzer cannot determine a clock domain for the node, or the clock domain is ambiguous. For example, the PowerPlay 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 PowerPlay Power Analyzer substitutes and reports a toggle rate of zero.

Vectorless Estimation

For some device families, the PowerPlay 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 PowerPlay Power Analyzer requires simulation data for at least the registered nodes and I/O nodes for accuracy.

The **PowerPlay 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 Quartus Prime software, on the Assignments menu, click **Settings**.
2. In the Category list, select **PowerPlay Power Analyzer Settings**.
3. Turn off the **Use vectorless estimation** option.

Using the PowerPlay Power Analyzer

For flows that use the PowerPlay 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.

Common Analysis Flows

You can use the analysis flows in this section with the PowerPlay Power Analyzer. However, vectorless activity estimation is only available for some device families.

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.

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.

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.

Signal Activities from User Defaults Only

The user defaults only flow provides the lowest degree of accuracy.

Importance of .vcd

Altera recommends that you use a **.vcd** or a **.saf** generated by gate-level timing simulation for an accurate power estimation because gate-level timing simulation takes all the routing resources and the exact logic array resource usage into account.

Generating a .vcd

In previous versions of the Quartus Prime software, you could use either the Quartus Prime simulator or an EDA simulator to perform your simulation. The 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 PowerPlay 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 PowerPlay 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. Altera 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 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.

Generating a .vcd from ModelSim Software

To generate a **.vcd** with the ModelSim software, follow these steps:

1. In the 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 step 6 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**.

Generating a .vcd from Full Post-Fit Netlist (Zero Delay) Simulation

To successfully generate a **.vcd** from the full post-fit Netlist (zero delay) simulation, follow these steps:

1. Compile your design in the Quartus Prime software to generate the Netlist `<project_name>`**.vo**.
2. In `<project_name>`**.vo**, search for the include statement for `<project_name>`**.sdo**, comment the statement out, and save the file.
   Altera recommends that you use the Standard Delay Format Output File (**.sdo**) for gate-level timing simulation. The **.sdo** contains the delay information of each architecture primitive and routing element in your design; however, you must exclude the **.sdo** for zero delay simulation.
3. Generate a .vcd for power estimation by performing the steps in **Generating a .vcd** on page 4-17.

PowerPlay Power Analyzer Compilation Report

The following table list the items in the Compilation Report of the PowerPlay 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 PowerPlay Power Analyzer Confidence Metric report.
Settings	The Settings section of the report shows the PowerPlay 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 **PowerPlay Power Analyzer Settings** page.
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.

Section	Description
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 PowerPlay 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.

Section	Description
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 **PowerPlay Power Analyzer Settings** page. Altera 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 Quartus Prime software generates during the analysis.

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 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 Information

- Tcl Scripting
- API Functions for Tcl

- **Quartus Prime Settings File Reference Manual**
- **Command-Line Scripting**

Running the PowerPlay Power Analyzer from the Command–Line

The executable to run the PowerPlay 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 Quartus Prime project called *sample*.

To instruct the PowerPlay Power Analyzer to generate a PowerPlay EPE File:

```
quartus_pow sample --output_epe=sample.csv ↵
```

To instruct the PowerPlay Power Analyzer to generate a PowerPlay EPE File without performing the power estimate:

```
quartus_pow sample --output_epe=sample.csv --estimate_power=off  ↵
```

To instruct the PowerPlay Power Analyzer to use a .vcd as input (sample.vcd):

```
quartus_pow sample --input_vcd=sample.vcd ↵
```

To instruct the PowerPlay 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 PowerPlay 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 **PowerPlay Power Analyzer Settings Operating Conditions** page. Use the 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 **PowerPlay Power Analyzer Compilation Report** on page 4-18.

Document Revision History

The following table lists the revision history for this chapter.

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *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 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 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.

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 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.

System Debugging Tools Overview 5

2015.11.02

QPP5V3 Subscribe Send Feedback

About Altera System Debugging Tools

The Altera® 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 manual 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.

System Debugging Tools Portfolio

The Quartus Prime software provides a portfolio of system design debugging tools for real-time verification of your design. Each tool in the system debugging portfolio uses a combination of available memory, logic, and routing resources to assist in the debugging process. The tools provide visibility by routing (or "tapping") signals in your design to debugging logic. The debugging logic is then compiled with your design and downloaded into the FPGA or CPLD for analysis. Because different designs can have different constraints and requirements, such as the number of spare pins available or the amount of logic or memory resources remaining in the physical device, you can choose a tool from the available debugging tools that matches the specific requirements for your design.

System Debugging Tools Comparison

Table 5-1: Debugging Tools Portfolio

Tool	Description	Typical Usage
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 provides real-time in-system debugging capabilities. Using System Console, you can read from and write to Memory Mapped components in our system without the help of a processor or additional software. System Console uses Tcl as the fundamental infrastructure which means 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 target FPGA you wish to debug.
Transceiver Toolkit	The Transceiver Toolkit allows you to test and tune transceiver link signal quality. You can use a combination of bit error rate (BER), bathtub curve, and eye contour graphs as quality 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.
SignalTap ® II Logic Analyzer	This logic analyzer uses FPGA resources to sample test nodes and outputs the information to the Quartus Prime software for display and analysis.	You have spare on-chip memory and you want functional verification of your design running in hardware.

Tool	Description	Typical Usage
SignalProbe	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 tool 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.

Altera JTAG Interface (AJI)

With the exception of SignalProbe, each of the on-chip debugging tools uses the JTAG port to control and read back data from debugging logic and signals under test. System Console uses JTAG and other interfaces as well. The JTAG resource is shared among all of the on-chip debugging tools.

Required Arbitration Logic

For all system debugging tools except System Console, the Quartus Prime software compiles logic into your design automatically to distinguish between data and control information and each of the debugging logic blocks, when the JTAG resource is required. This arbitration logic, also known as the System-Level Debugging (SLD) infrastructure, is shown in the design hierarchy of your compiled project as

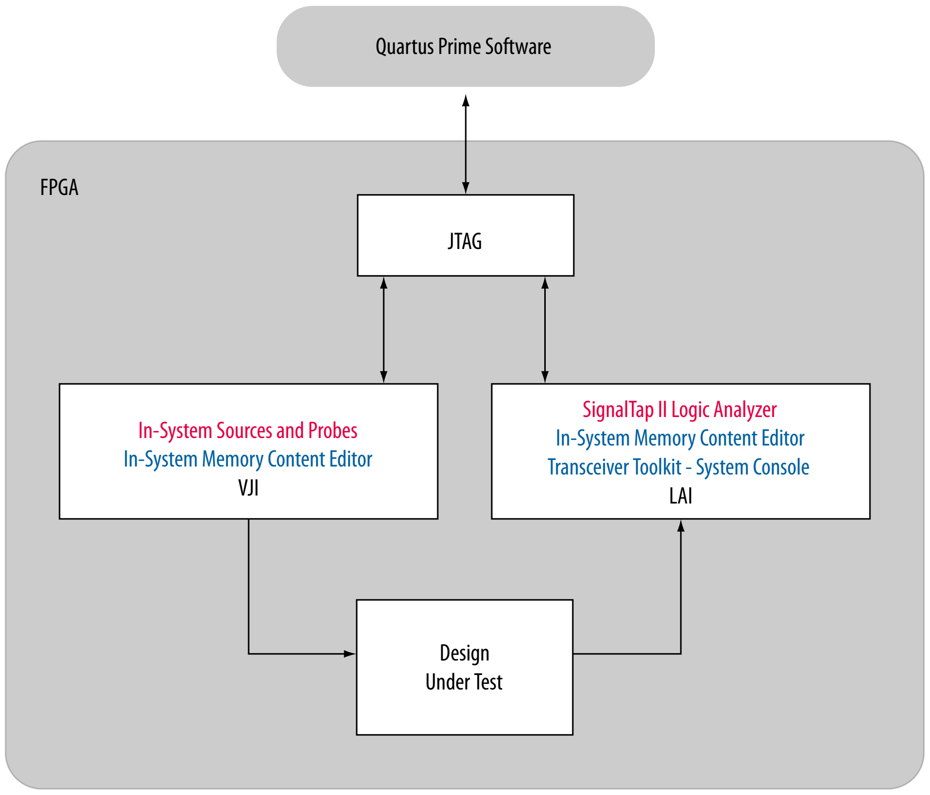
sld_hub:sld_hub_inst. The SLD logic allows you to instantiate multiple debugging blocks into your design and run them simultaneously. For System Console, you must explicitly insert debug IP cores into your design to enable debugging.

Debugging Ecosystem

To maximize debugging closure, the Quartus Prime software allows you to use a combination of the debugging tools in tandem to fully exercise and analyze the logic under test. All of the tools have basic analysis features built in; that is, all of the tools enable you to read back information collected from the design nodes that are connected to the debugging logic. Out of the set of debugging tools, the SignalTap II Logic Analyzer, the LAI, and the SignalProbe feature are general purpose debugging 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 the debugging breakpoints, 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 debugging solution.

Figure 5-1: Debugging Ecosystem


Quartus Prime Software
FPGA
JTAG
In-System Sources and Probes
In-System Memory Content Editor
VJI
SignalTap II Logic Analyzer
In-System Memory Content Editor
Transceiver Toolkit - System Console
LAI
Design Under Test

About Analysis Tools for RTL Nodes

The SignalTap II Logic Analyzer, SignalProbe, and LAI are designed specifically for probing and debugging RTL signals at system speed. They are general-purpose analysis tools that enable you to tap and analyze any routable node from the FPGA or CPLD. If you have spare logic and memory resources, the

SignalTap II Logic Analyzer is useful for providing fast functional verification of your design running on actual hardware.

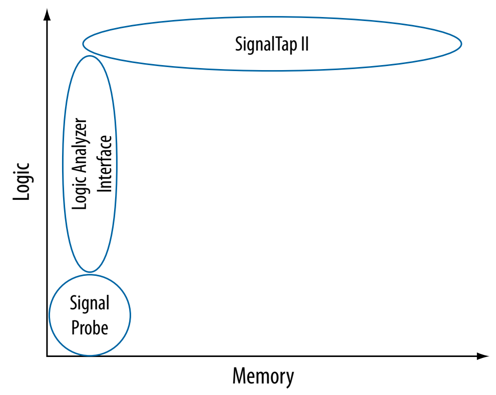
Conversely, if logic and memory resources are tight and you require the large sample depths associated with external logic analyzers, both the LAI and the SignalProbe make it easy to view internal design signals using external equipment.

Note: The SignalTap II Logic Analyzer is not supported on CPLDs, because there are no memory resources available on these devices.

Resource Usage

The most important selection criteria for these three tools are the available resources remaining on your device after implementing your design and the number of spare pins available. You should evaluate your preferred debugging option early on in the design planning process to ensure that your board, your Quartus Prime project, and your design are all set up to support the appropriate options. Planning early can reduce time spent during debugging and eliminate the necessary late changes to accommodate your preferred debugging methodologies.

Figure 5-2: Resource Usage per Debugging Tool



Overhead Logic

Any debugging tool that requires the use of a JTAG connection requires the SLD infrastructure logic, for communication with the JTAG interface and arbitration between any 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. The overhead logic is shared between all available debugging modules in your design. Both the SignalTap II Logic Analyzer and the LAI use a JTAG connection.

For SignalProbe

SignalProbe requires very few on-chip resources. Because it requires no JTAG connection, SignalProbe uses no logic or memory resources. SignalProbe 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.

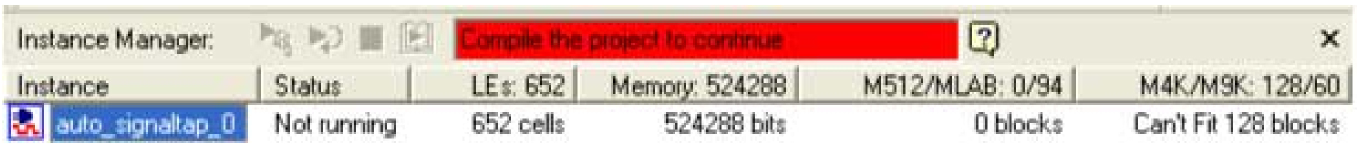
For SignalTap II

The SignalTap II 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 SignalTap II 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 SignalTap II Logic Analyzer instance to capture data and the sample depth that your design requires for debugging. For the SignalTap II Logic Analyzer, there is the added benefit of requiring no external equipment, as all of the triggering logic and storage is on the chip.

Resource Estimation

The resource estimation feature for the SignalTap II 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 5-3: Resource Estimator



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

Pin Usage

For SignalProbe

The ratio of the number of pins used to the number of signals tapped for the SignalProbe 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.

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.

For SignalTap II

Other than the JTAG test pins, the SignalTap II Logic Analyzer uses no additional pins. All data is buffered using on-chip memory and communicated to the SignalTap II Logic Analyzer GUI via the JTAG test port.

Usability Enhancements

The SignalTap II Logic Analyzer, SignalProbe, 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. SignalProbe inserts signals directly from your post-fit database. The SignalTap II Logic Analyzer and LAI support inserting signals from both pre-synthesis and post-fit netlists.

Incremental Routing

SignalProbe 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 SignalProbe, you can save as much as 90% compile time of a full compilation.

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 SignalTap II 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.

Remote Debugging

You can perform remote debugging of your system with the 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 Altera SoC to use the SLD tools to perform remote debugging.

Related Information

- Application Note 624: Debugging with System Console over TCP/IP
- Application Note 693: Remote Debugging over TCP/IP for Altera SoC

Suggested On-Chip Debugging Tools for Common Debugging Features

Table 5-2: Tools for Common Debugging Features [(1)]

Feature	SignalProbe	Logic Analyzer Interface (LAI)	SignalTap II 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 SignalTap II Logic Analyzer. No data is captured or stored with SignalProbe.

Feature	SignalProbe	Logic Analyzer Interface (LAI)	SignalTap II Logic Analyzer	Description
Ease in Debugging Timing Issue	X	X	—	External equipment, such as oscilloscopes and mixed signal oscilloscopes (MSOs), can be used with either LAI or SignalP-robe. 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 SignalTap II Logic Analyzer has little effect on the design, because it is set as a separate design partition. SignalProbe incrementally routes nodes to pins, not affecting the design at all.

Feature	SignalProbe	Logic Analyzer Interface (LAI)	SignalTap II Logic Analyzer	Description
Short Compile and Recompile Time	X	X *(2)*	X *(2)*	SignalProbe attaches incrementally routed signals to previously reserved pins, requiring very little recompilation time to make changes to source signal selections. The SignalTap II Logic Analyzer and the LAI can refit their own design partitions to decrease recompilation time.
Triggering Capability	N/A	N/A	X	The SignalTap II Logic Analyzer offers triggering capabilities that are comparable to commercial logic analyzers.
I/O Usage	—	—	X	No additional output pins are required with the SignalTap II Logic Analyzer. Both the LAI and SignalProbe require I/O pin assignments.

Feature	SignalProbe	Logic Analyzer Interface (LAI)	SignalTap II Logic Analyzer	Description
Acquisition Speed	N/A	—	X	The SignalTap II 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 Quartus Prime software. SignalProbe and SignalTap II do not require a host for debugging purposes.

Feature	SignalProbe	Logic Analyzer Interface (LAI)	SignalTap II Logic Analyzer	Description
No External Equipment Required	—	—	X	The SignalTap II 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 Quartus Prime software or the stand-alone SignalTap II Logic Analyzer software. SignalP-robe 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.

About 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 all three tools can be used to achieve the same results, there are some considerations that make one tool easier to use in certain applications than others. 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 more 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 have instantiated into your design.

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.

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.

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.

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.

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.

System Console

System Console is a framework that you can launch from the Quartus Prime software to start services for performing various debugging tasks. System Console provides you with Tcl scripts and a GUI to access the Qsys 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 Qsys 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.

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.

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 Altera SoC processor, as well as access modules that produce or consume a stream of bytes.

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 then automatically run EyeQ and Auto Sweep testing to view a graphical representation of your test data.

Document Revision History

Table 5-3: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *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.
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 Information
Quartus Handbook Archive
For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.

Analyzing and Debugging Designs with System Console 6

2015.11.02

QPP5V3 Subscribe Send Feedback

Introduction to System Console

System Console provides visibility into your design and allows you to perform low-level debugging on an FPGA running in real-time. System Console performs tests on debug-enabled Qsys instantiated IP cores. A variety of debug services provide read and write access to elements in you design to facilitate debugging.

You can perform the following tasks with System Console and the tools built on top of System Console:

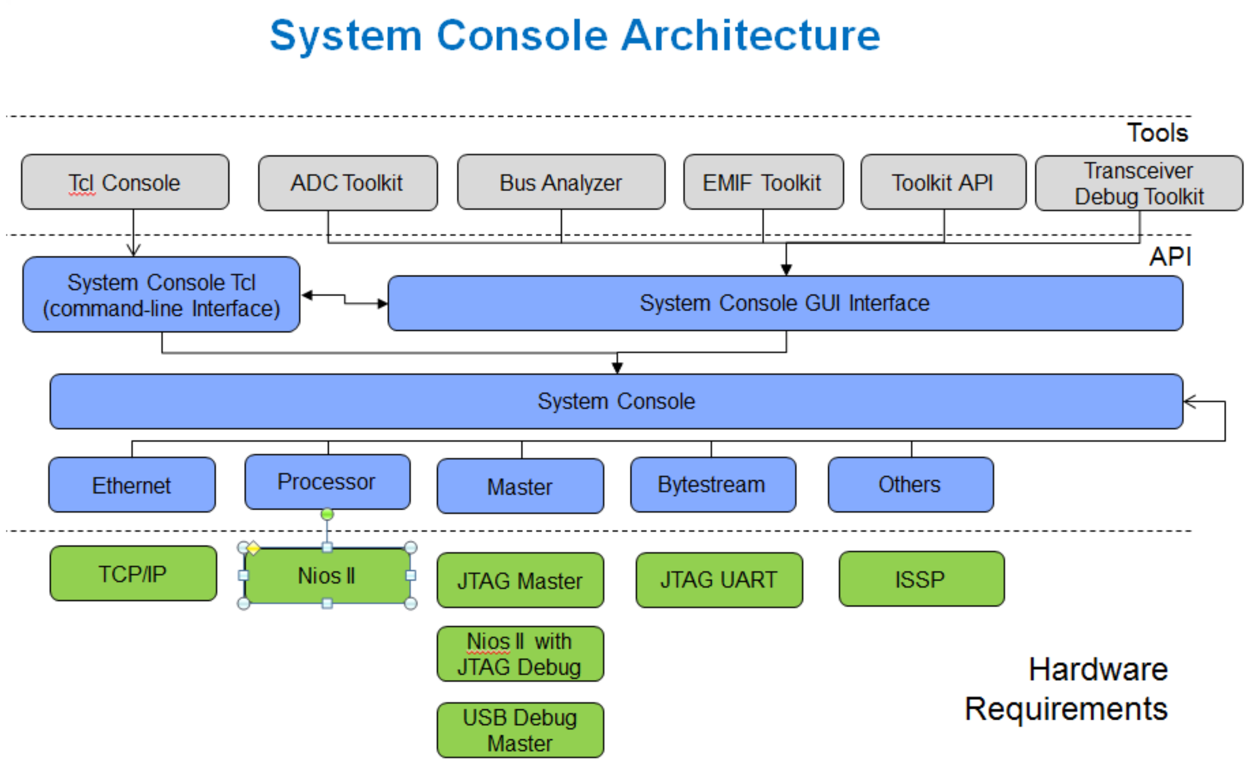
- Bring up boards with both finalized and partially complete designs.
- Perform remote debug with internet access.
- Automate run-time verification through scripting across multiple devices in your system.
- Test serial links with point-and-click configuration tuning in the Transceiver Toolkit.
- Debug 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 MAX® 10 device with the ADC Toolkit.
- Perform system verification with MATLABS/Simulink.

In the diagram below, tools are graphical interface, such as Bus Analyzer, that you use in conjunction with the System Console GUI interface.

Tcl Console is a command-line interface that provides you access to the System Console core. API supports services and enables communication with hardware, such as, Ethernet, processor, master, and bytestream services.

Some service have hardware requirements, for example the master service requires a JTAG Master or Nios II with JTAG Debug, or simular. The Bytestream service requires a JTAG UART.

Note: You use debug links to connect the host to the target that you are debugging.



Related Information

- External Memory Interface Documentation
- Debugging Transceiver Links Documentation on page 7-1
- Application Note 693: Remote Hardware Debugging over TCP/IP for Altera 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

Hardware Requirements for 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 Qsys IP Catalog. Some IP cores are enabled for debug by default, while others are enabled for debug with options in the parameter editor. You can use some debug agents, such as the Nios II processor with debug enabled, for multiple purposes. For example, debugging the hardware in your design, as well as debugging the code running on the Nios II processor.

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. For example, when you add a JTAG to Avalon Master Bridge instance to a Qsys system, you can read and write to memory-mapped slaves connected to the bridge.

Services associated with debug agents in the running design can be directly opened and closed as needed for debugging. System Console is responsible for determining and using 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 System Console with channel communication information. When System Console opens in the Quartus Prime software or Qsys while your design is open, any existing **.sof** is automatically found and linked to the detected running device. You can may need to link the design and device manuallly in a complex system.

Note: The following IP cores in the IP Catalog do not support VHDL simulation generation in the current version of the Quartus Prime software:

- JTAG Debug Link
- SLD Hub Controller System
- USB Debug Link

Related Information

WP-01170 System-Level Debugging and Monitoring of FPGA Designs

IP Cores that Interact with System Console

To facilitate debugging your design with the System Console, you can include IP cores with debug interfaces that the System Console can use to send commands and receive data. By adding the appropriate debug agent to your design, System Console services can use the associated capabilities of the debug agent.

Table 6-1: 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

Service	Function	Debug Agent Providing Service
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.	• NiosII with debug • JTAG to Avalon Master Bridge • USB Debug Master 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

Related Information

- **System Console Examples and Tutorials** on page 6-77
- **System Console Commands** on page 6-8

System Console Flow

1. Add an IP Core to your Qsys system.
2. Generate your Qsys system.
3. Compile your design in the Quartus Prime software.
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.

Starting System Console

In the System Console, you use Tcl scripting commands to interact with your design in both graphical and command-line interface modes. The System Console GUI panes provide design information for debugging your design.

Related Information

- **System Console Commands** on page 6-8
- **Scripting Reference Manual**
  Information about Tcl scripting support
- **Introduction to Tcl Online Training**

Customizing Startup

You can customize your System Console environment, as follows:

- Adding 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.
- Specifying 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.
- Using 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.

Starting System Console from Nios II Command Shell

1. On the Windows Start menu, click **All Programs** > **Altera** > **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.

Starting System Console Stand-Alone

You can get the stand-alone version of System Console as part of the Quartus Prime software Programmer and Tools installer on the Altera website.

1. Navigate to the **Altera Download Center** page and click the **Additional Software** tab.
2. On the Windows Start menu, click **All Programs** > **Altera <version>** > **Programmer and Tools** > **System Console**

Related Information

- Altera Download Center

Starting System Console from Qsys

- Click **Tools** > **System Console**.

Starting System Console from

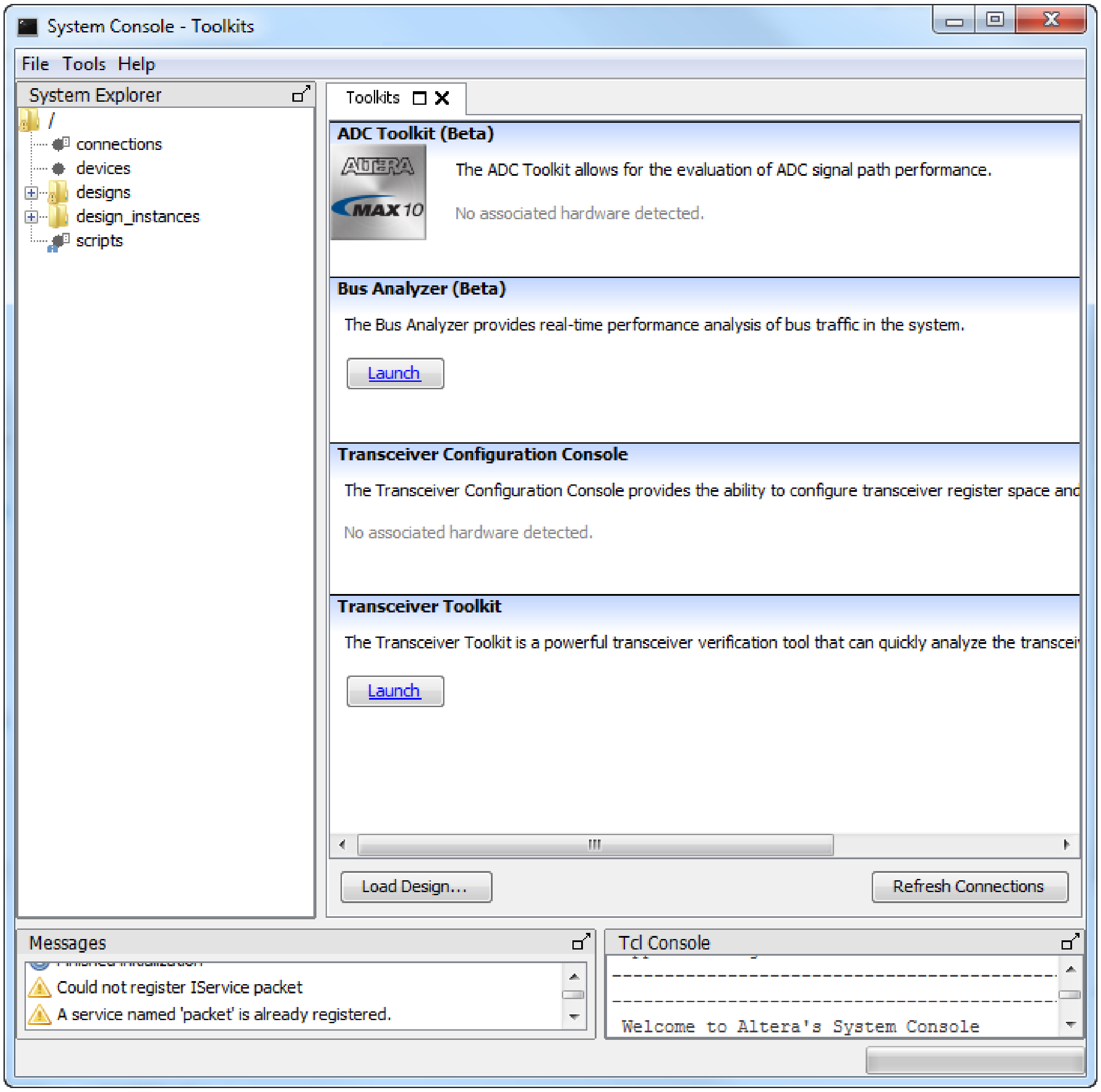
- Click **Tools** > **System Debugging Tools** > **System Console**.

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.
- **Toolkits**—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 6-1: 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
```

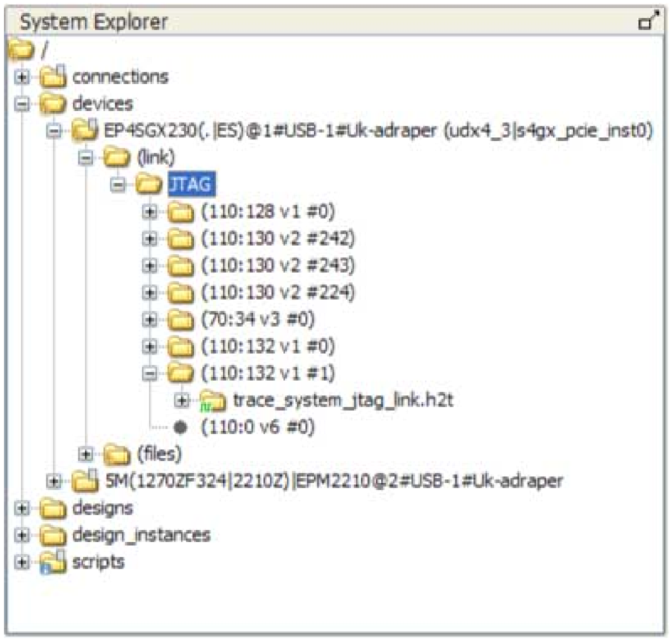
System Explorer Pane

The **System Explorer** pane displays the virtual file system for all connected debugging IP cores, and contains the following information:

- **Devices** folder—Contains information about each device connected to System Console.
- **Scripts** folder—Stores scripts for easy execution.
- **Connections** folder—Displays information about the board connections which are visible to System Console, such as USB Blaster. Multiple connections are possible.
- **Designs** folder—Displays information about project designs connected to System Console. Each design represents a **.sof** file that is loaded.

The **Devices** contains a sub-folder for each device currently connected to System Console. Each device sub-folder contains a **(link)** folder and sometimes contains 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 project for the device.

Figure 6-2: System Explorer Pane



```
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
```

- The figure above 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.
- 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 above 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 above has a running clock.

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 6-2: 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.

Command	Arguments	Function
`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.

Command	Arguments	Function
`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. 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 Altera 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.

Command	Arguments	Function
`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 Information

- **Starting System Console** on page 6-4
- **Remote Hardware Debugging over TCP/IP for Altera SoC**

Running System Console in Command-Line Mode

You can run System Console in command line mode and work interactively or run a Tcl script. The System Console prints responses to your commands 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.

Locating, Opening, and Closing System Console Services

System Console services allow you to access different parts of your running design. For example, the services you can access to memory-mapped slave interfaces. With processor services, you can access embedded processor controls. Services do not inter-mix, but a single IP core can provide multiple services. For example, the Nios II processor contains a debug core and has a memory-mapped master interface that can connect to slaves. The master service can access the memory-mapped slaves that connect to the Nios II processor. You can also use the processor service to debug your design.

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. You can retrieve service paths for a particular service with the command `get_service_paths` <*service-type*>.

Example 6-1: 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 `get_service_paths` to obtain service paths rather than hard coding them into your Tcl scripts.

The string values of service paths change with different releases of the tool, so you should not infer meaning from the actual strings within the service path. Use `marker_node_info` 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 6-2: Marker_node_info

You can also use the `marker_node_info` command to get information about the discovered services so you can choose the right one.

```
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)
```

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 6-3: 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 6-4: Closing a Service

```
close_service master $claim_path; #Closes the service.
```

System Console Services

Altera's System Console services provide access to hardware modules instantiated in your FPGA. Services vary in the type of debug access they provide.

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 6-5: 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 Information

- **SLD Commands** on page 6-14
- **Virtual JTAG Megafunction documentation**

SLD Commands

Table 6-3: 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.

Related Information

SLD Service on page 6-13

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 Quartus Prime software.

Example 6-6: 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 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 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
```

Related Information

In-System Sources and Probes Commands on page 6-16

In-System Sources and Probes Commands

Note: The valid values for ISSP claims include `read_only`, `normal`, and `exclusive`.

Table 6-4: 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`
		`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.

Related Information

In-System Sources and Probes Service on page 6-15

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 6-7: 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.

Related Information

Monitor Commands on page 6-18

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 6-5: 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.

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 6-6: 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`.

Command	Arguments	Function
`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.

Related Information

Monitor Service on page 6-16

Device Service

The device service supports device-level actions.

Example 6-8: 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
```

Related Information

Device Commands on page 6-21

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 6-7: 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.

Related Information

Device Service on page 6-20

Design Service

You can use design service commands to work with design information.

Example 6-9: Load

When you open System Console from the software or Qsys, 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 6-10: 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**.

Related Information

Design Service Commands on page 6-22

Design Service Commands

Design service commands load and work with your design at a system level.

Table 6-8: Design Service Commands

Command	Arguments	Function
`design_load`	*<quartus-project-path>*, *<sof-file-path>*, or *<qpf-file-path>*	Loads a model of a design into System Console. Returns the design path. For example, if your 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 logical design with a physical device. For example, you can link a 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 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 *Altera 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.

Related Information

Design Service on page 6-21

Bytestream Service

The bytestream service provides access to modules that produce or consume a stream of bytes. You can 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 6-11: 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
```

Related Information

Bytestream Commands on page 6-23

Bytestream Commands

Table 6-9: 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.

Related Information

Bytestream Service on page 6-23

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]
```

JTAG Debug Commands

JTAG Debug commands help debug the JTAG Chain connected to a device.

Table 6-10: 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.

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.

Related Information

Verifying Clock and Reset Signals on page 6-79

Working with Toolkits

You can use Toolkit API to create tools to visualize and interact with design debug data.

Graphical widgets in the form of buttons and text fields can leverage user input to interact with debug logic. Toolkit API is the successor to the Dashboard service. Use Toolkit API with the Quartus Prime software versions 14.1 and later.

Toolkits require the following files:

- XML file that describes the plugin (**.toolkit** file)
- Tcl file that implements the toolkit GUI

Registering a Toolkit

Use the `toolkit_register` command to register a System Console toolkit. You must also specify the path to the **.toolkit** file when you use the `toolkit_register` command.

When you open System Console, toolkits that appear with the **.toolkit** extention in the **$HOME/system_console/toolkits/** directory appear in the **Tools** > **Toolkits** menu.

Opening a Toolkit

You can use the **Toolkits** tab in System Console to launch available toolkits. Each toolkit has a description, detected hardware list, and a launch button.

To launch a toolkit from the Quartus Prime software:

1. Click **Tools** > **System Debugging Tools** > **System Console**.
2. In System Console, to view available toolkits, click **Tools** > **Toolkits**.

You can also use following command-line command to open a System Console toolkit:

`toolkit_open` <*toolkit_name*>, where <*toolkit_name*> is the name in the **.toolkit** file.

Note: 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>` with the `set context [toolkit_get_context]`command.

Creating a Toolkit Description File

The Toolkit Description File (**.toolkit**) provides the registration data for a toolkit. It does not create an instance of the toolkit GUI.

The toolkit file includes the following attributes:

- Unique **<*toolkit name*>.toolkit** name that describes the toolkit type.
- Internal toolkit file name.
- Toolkit display name that appears in the GUI.
- Whether the System Console **Tools** > **Toolkits Tools** menu displays the toolkit.
- Path to the **.tcl** file that implements the toolkit.
- Description of the purpose of the toolkit (optional).
- Path to an icon to display as the toolkit launcher button in System Console (optional).

  Note: The **.png** format 64x64 is preferred. If the icon does not take up the whole space, then be sure that the background is transparent.
- Requirement is a toolkit that displays as associated hardware for the toolkit (optional).

  Note: If this toolkit works with a particular type of hardware, specify the debug type name of the hardware with the `requirement` property. This enables automatic discovery of the toolkit. The debug type name of a toolkit is the name of the `hw.tcl` component, a dot, and the name of the interface within that component which the toolkit will use, for example, **<hw.tcl name><interface name>**.

Related Information

- **Toolkit API GUI Example .toolkit File** on page 6-32
- **Toolkit API GUI Example .tcl File** on page 6-32

Matching Toolkits with IP Cores

You can match any toolkit with any IP core.

- When the toolkit searches for IP, it 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
  ```
- You can 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.

Toolkit API

The Toolkit API service enables you to construct GUIs for visualizing and interacting with debug data. Toolkit API is a graphical pane for the layout of your graphical widgets, which can include buttons and text fields. Widgets can pull data from other System Console services. Similarly, widgets can leverage user input to act on debug logic in your design through services.

Properties

Widget properties can push and pull information to the user interface. Widgets have properties specific to their type. For example, the button property `onClick` performs an action when the button is clicked. A label widget does not have the same property because it does not perform an action when clicked. 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 can implement group-type widgets that contain child widgets. Layout properties dictate layout actions performed by a parent on its children.

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. You 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 can 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 a button is clicked. Callbacks may update widgets or interact with services based on the contents of a text field, or the state of any other widget.

Customizing Toolkit API Widgets

Use `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.

Toolkit API Script Examples

Note: To convert your Dashboard scripts to Toolkit API, do the following:

1. Add a **.toolkit** file.
2. Remove the `add_service dashboard <`*`name of service`*`>` command.
3. Change `dashboard_<`*`command`*`>` to `toolkit_<`*`command`*`>`.
4. Change `open_service` to `claim_service`, for example:

   Before:

   ```
   open_service slave $path
   master_read_memory $path address count
   ```

After:

```
set c [claim_service slave $path lib {}]
master_read_memory $c address count
```

Example 6-12: Registering the Service

Toolkit API is not initialized by default. You must register the service before you can use it.

```
toolkit_register <toolkit_file>
```

Example 6-13: Making the Toolkit Visible in System Console

Once you register a toolkit, you must explicitly make the toolkit visible. Use the `toolkit_set_property` command to modify the `visible` property of the root toolkit.

```
toolkit_set_property $toolkit root_widget visible true
```

In this command, `$toolkit` represents the Toolkit API service. `root_widget` is the name of the root toolkit widget. `visible` is the property that allows the toolkit to be visible in System Console.

Example 6-14: Adding Widgets

Use the `toolkit_add` command to add widgets.

```
toolkit_add my_button button "parentGroup"
```

Use the following commands to add a label widget "my_label" to the root toolkit. In the GUI, it appears as "Widget Label."

```
set content "Text to display goes here"
toolkit_set_property $dash $name text $content
```

This command sets the `text` property to that string. In the GUI, the displayed text changes to the new value. Add one more label:

```
toolkit_add $dash my_label_2 label self
toolkit_set_property $dash my_label_2 text "Another label"
```

Notice the new label appears to the right of the first label. Cause the layout to put the label below instead:

```
toolkit_set_property $dash self itemsPerRow 1
```

Example 6-15: Gathering Input

Incorporate user input into your Toolkit API:

```
set name "my_text_field"
set widget_type "textField"
set parent "self"
toolkit_add $dash $name $widget_type $parent
```

The widget appears, but it is very small. Make the widget fill the horizontal space:

```
toolkit_set_property $dash my_text_field expandableX true
```

Now the text field is fully visible. Text can be typed into it once clicked. Type a sentence. Now, retrieve the contents of the field:

```
set content [toolkit_get_property $dash my_text_field text]
puts $content
```

This prints the contents into the console.

Example 6-16: Updating Widgets Upon User Events

You can make Toolkit API perform actions without interactive typing. Use callbacks to accomplish this. Start by defining a procedure that updates the first label with the text field contents:

```
proc update_my_label_with_my_text_field {dash} {
    set content [toolkit_get_property $dash my_text_field text]
    toolkit_set_property $dash my_label text $content
}
```

Run the `update_my_label_with_my_text_field $dash` command in the Tcl Console. Notice that the first label now matches the text field contents. Have the `update_my_label_with_my_text_field $dash` command called whenever the text field changes:

```
toolkit_set_property $dash my_text_field onChange \
"update_my_label_with_my_text_field $dash"
```

The `onChange` property is executed each time the text field changes. The effect is the first field changes to match what is typed.

Example 6-17: Buttons

You can use buttons to trigger actions. Create a button that changes the second label:

```
proc append_to_my_label_2 {dash suffix} {
    set old_text [toolkit_get_property $dash my_label_2 text]
    set new_text "${old_text}${suffix}"
    toolkit_set_property $dash my_label_2 text $new_text
}
set text_to_append ", and more"
toolkit_add $dash my_button button self
toolkit_set_property $dash my_button onClick [list append_to_my_label_2 \
$dash $text_to_append]
```

Click the button and the second label gets some text appended to it.

Example 6-18: Groups

The property `itemsPerRow` dictates how widgets are laid out in a group. For more complicated layouts where the number of widgets per row is different per row, use nested groups. Add a new group with more widgets per row:

```
toolkit_add $dash my_inner_group group self
toolkit_set_property $dash my_inner_group itemsPerRow 2
toolkit_add $dash inner_button_1 button my_inner_group
toolkit_add $dash inner_button_2 button my_inner_group
```

There is now a row with a group of two buttons. You can remove the border with the group name to make the nested group more seamless.

```
toolkit_set_property $dash inner_group title ""
```

The `title` property can be set to any other string to have the border and title text show up.

Example 6-19: Tabs

GUIs do not require all the widgets to be visible at the same time. Tabs accomplish this.

```
toolkit_add $dash my_tabs tabbedGroup self
toolkit_set_property $dash my_tabs expandableX true
toolkit_add $dash my_tab_1 group my_tabs
toolkit_add $dash my_tab_2 group my_tabs
toolkit_add $dash tabbed_label_1 label my_tab_1
toolkit_add $dash tabbed_label_2 label my_tab_2
toolkit_set_property $dash tabbed_label_1 text "in the first tab"
toolkit_set_property $dash tabbed_label_2 text "in the second tab"
```

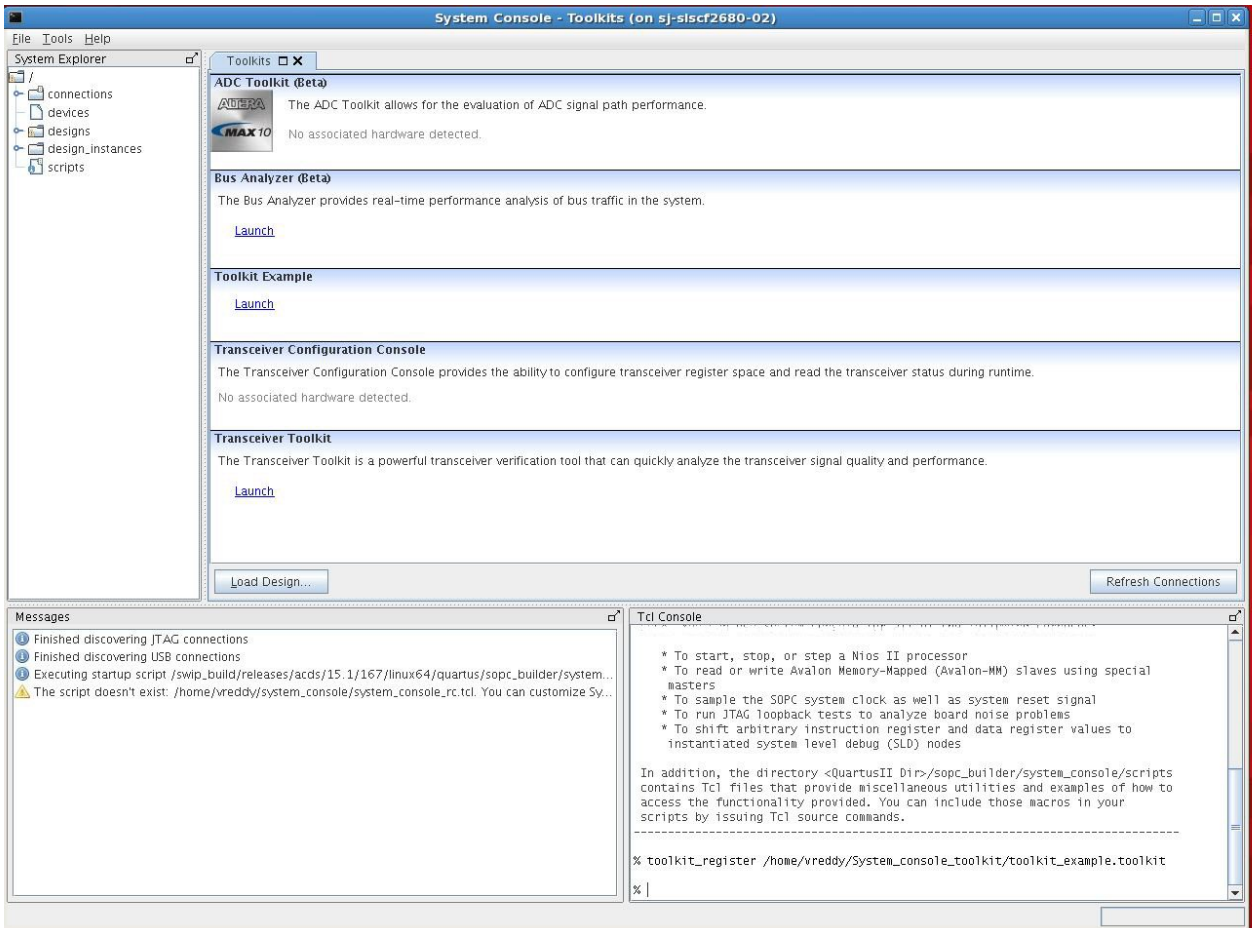
This adds a set of two tabs, each with a group containing a label. Clicking on the tabs changes the displayed group/label.

Toolkit API GUI Example

The Toolkit API GUI example creates a button in the **Toolkits** pane in the System Console window, which registers the toolkit. Clicking the **Launch** button under **Toolkit Example** opens a GUI window that provides debug interaction with your design.

The Toolkit Example includes the **.toolkit** and **.tcl** files so that you can reconstruct the example on your local system.

Register Toolkit API in System Console



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 Configuration Console provides the ability to configure transceiver register space and read the transceiver status during runtime.

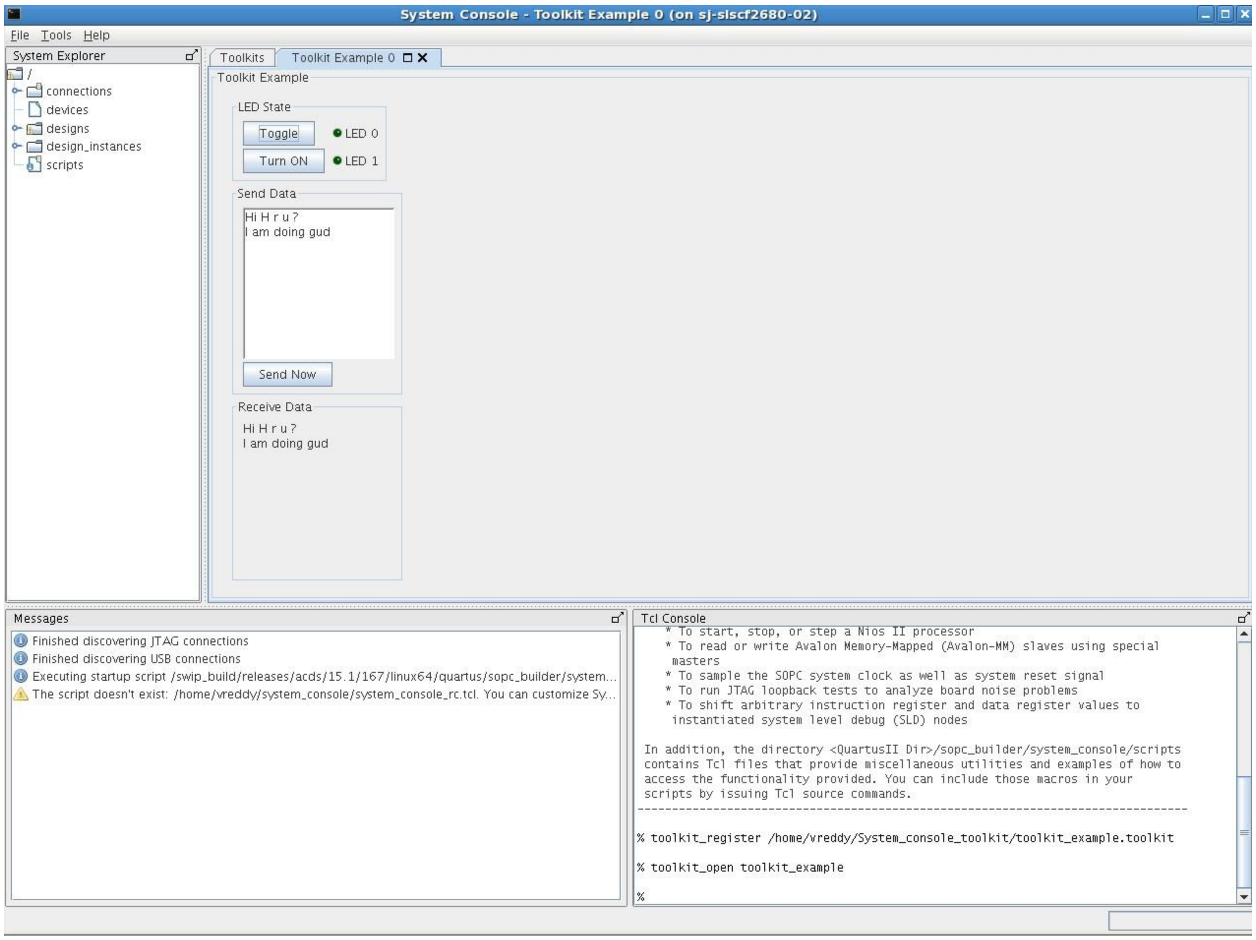
No associated hardware detected.

The Transceiver Toolkit is a powerful transceiver verification tool that can quickly analyze the transceiver signal quality and performance.

```
Finished discovering JTAG connections
Finished discovering USB connections
Executing startup script /swip_build/releases/acds/15.1/167/linux64/quartus/sopc_builder/system...
The script doesn't exist: /home/vreddy/system_console/system_console_rc.tcl. You can customize Sy...
```

```
    * To start, stop, or step a Nios II processor
    * To read or write Avalon Memory-Mapped (Avalon-MM) slaves using special
     masters
    * 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 /home/vreddy/System_console_toolkit/toolkit_example.toolkit

% |
```

Toolkit API GUI Example



Hi H r u ?
I am doing gud

Hi H r u ?
I am doing gud

```
Finished discovering JTAG connections
Finished discovering USB connections
Executing startup script /swip_build/releases/acds/15.1/167/linux64/quartus/sopc_builder/system...
The script doesn't exist: /home/vreddy/system_console/system_console_rc.tcl. You can customize Sy...
```

```
   * To start, stop, or step a Nios II processor
   * To read or write Avalon Memory-Mapped (Avalon-MM) slaves using special
    masters
   * 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 /home/vreddy/System_console_toolkit/toolkit_example.toolkit

% toolkit_open toolkit_example

%
```

Toolkit API GUI Example .toolkit File

The .toolkit files registers the Toolkit API to appear in the System Console.

```
<?xml version="1.0" encoding="UTF-8"?>
<toolkit name="toolkit_example" displayName="Toolkit Example" addMenuItem="true">
    <file> toolkit_example.tcl </file>
</toolkit>
```

Related Information

Creating a Toolkit Description File on page 6-26

Toolkit API GUI Example .tcl File

The Toolkit API .tcl file creates the 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 expandableY 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 expandableY 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 200
            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 expandableY 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 200
            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
```

Related Information

Creating a Toolkit Description File on page 6-26

Toolkit API Commands

Toolkit API commands run in the context of a unique toolkit instance.

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 /data/trogdor/toolkits/burninator.xml
```

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
```

get_quartus_ini

Description

Returns a value from the software **.ini** file.

Usage

`get_quartus_ini` *<ini> <type>*

Returns

No return value.

Arguments

<ini>

Name of the 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]
```

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

Returns the context.

Arguments

None

N/A

Example

```
set context [toolkit_get_context]
```

toolkit_get_types

Description

Returns a list of widget types.

Usage

```
toolkit_get_types
```

Returns

No return value.

Arguments

None

N/A

Example

```
set widget_names [toolkit_get_types]
```

toolkit_get_properties

Description

Returns a list of toolkit properties for a type of widget.

Usage

`toolkit_get_properties` *<widgetType>*

Returns

No return value.

Arguments

<widgetType>

Name of a type of widget.

Example

```
set widget_properties [toolkit_get_properties xyChart]
```

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.

Example

```
toolkit_add my_button button "parentGroup"
```

toolkit_get_property

Description

Returns the value of a property for a specific widget.

Usage

`toolkit_get_property` *<id> <propertyName>*

Returns

No return value.

Arguments

<id>

A unique ID for the widget being queried.

<tpropertyName>

The name of the widget property being queried.

Example

```
set enabled [toolkit_get_property my_button enabled]
```

toolkit_set_property

Description

Sets the value of a property 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 false
```

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
```

toolkit_get_widget_dimensions

Description

Returns the width and height of the specified widget.

Usage

`toolkit_get_widget_dimensions` *<id>*

Returns

No return value.

Arguments

<id>

A unique ID for the widget being added.

Example

```
set dimensions [toolkit_get_widget_dimensions my_button]
```

Toolkit API Properties

Widget Types and Properties

Table 6-11: Toolkit API Widget Types and Properties

Name	Description
`enabled`	Enables or disables the widget.
`expandable`	Allows the widget to be expanded.
`expandableX`	Allows the widget to be resized horizontally if there is space available in the cell where it resides.
`expandableY`	Allows the widget to be resized 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.

barChart Properties

Table 6-12: 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. Range is specified in a Tcl list, for example: `[list lower_numerical_value upper_numerical_value]`
`itemValue`	Value is specified in a Tcl list, for example: `[list bar_category_str numerical_value]`

button Properties

Table 6-13: Toolkit API button Properties

Name	Description
`onClick`	A Tcl command to run, usually a `proc`, every time the button is clicked.
`text`	The text on the button.

checkBox Properties

Table 6-14: Toolkit API checkBox Properties

Name	Description
`checked`	If true, the checkbox is checked. If false, the checkbox is not checked.
`onClick`	A Tcl command to run, usually a proc, every time the checkbox is clicked.
`text`	The text on the checkbox.

comboBox Properties

Table 6-15: Toolkit API comboBox Properties

Name	Description
`onChange`	A Tcl callback to run when the value of the combo box changes.
`options`	A list of options to display in the combo box.
`selected`	The index of the selected item in the combo box.

dial Properties

Table 6-16: 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 should mark. It must be between min and max.

fileChooserButton Properties

Table 6-17: Toolkit API fileChooserButton Properties

Name	Description
`text`	The text on the button.
`onChoose`	A Tcl command to run, usually a `proc`, every time the button is clicked.
`title`	The dialog box title.
`chooserButtonText`	chooserButtonText
`filter`	The file filter based on extension. Only one extension is supported. By default, all file names are allowed. The filter is specified as `[list filter_description file_extension]`, for example: `[list "Text Document (.txt)" "txt"]`
`mode`	Specifies what kind of files or directories can be selected. The default is `files_only`. Possible options are `files_only` and `directories_only`.
`multiSelectionEnabled`	Controls whether multiple files can be selected. False, by default.
`paths`	Returns a list of file paths selected in the file chooser dialog box. This property is read-only. It is most useful when used within the `onClick` script or a procedure when the result is freshly updated after the dialog box closes.

group Properties

Table 6-18: 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 number of widgets the group can position in one row, from left to right, before moving to the next row.

label Properties

Table 6-19: Toolkit API label Properties

Name	Description
`text`	The text to show in the label.

led Properties

Table 6-20: 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 color of the LED. The options are: `red_off`, `red`, `yellow_off`, `yellow`, `green_off`, `green`, `blue_off`, `blue`, and `black`.

lineChart Properties

Table 6-21: 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. You specify range in a Tcl list, for example: `[list lower_numerical_value upper_numerical_value]`
`Y-axis value range. By default, it is auto range. Range is specified in a Tcl list, for example [list lower_numerical_value upper_numerical_value].`	Y-axis value range. By default, it is auto range. You specify range in a Tcl list, for example: `[list lower_numerical_value upper_numerical_value]`

list Properties

Table 6-22: 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.

pieChart Properties

Table 6-23: Toolkit API pieChart Properties

Name	Description
`title`	Chart title.
`itemValue`	Item value. Value is specified in a Tcl list, for example: `[list bar_category_str numerical_value]`

tableProperties

Table 6-24: Toolkit API tableProperties

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 to be filled in the cell specified in the current `rowIndex` and `columnIndex` (Empty, by default).
`selectedRows`	Control or retrieve row selection.
`columnHeader`	The text to be filled 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 used. This is applicable only if `rowSorterEnabled` is `true`. Each column has its own sorting type. Supported types are `string` (default), `int`, and `float`.
`columnWidth`	The number of pixels used for 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 will be `{A1 B1 C1 A2 B2 C2 ...}`.

text Properties

Table 6-25: 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.

textField Properties

Table 6-26: Toolkit API textField Properties

Name	Description
`editable`	Controls whether the text box is editable.
`onChange`	A Tcl callback to run when the contents of the text box is changed.
`text`	The text to show in the text box.

timeChart Properties

Table 6-27: 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 is of the form `{low, high}`. The low/high values are interpreted as doubles.
`showLegend`	Sets whether a legend for the series should be shown in the graph.

xyChart Properties

Table 6-28: 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 interpreted as doubles.
`maximumItemCount`	Sets the maximum number of data values to keep in a series of data. Setting this does not affect the current data, only new data added to the chart. If a series is added that has more values than the maximum count, the last-maximumItemCount number of entries will be 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 most-recently used series data wins. If the identifier does not come with series data, that series is removed from the chart. The series is in a format such as the following: `{identifier, x-1 y-1, x-2 y-2}`.
`showLegend`	Sets whether a legend for the series should be shown in the graph.

ADC Toolkit

The ADC Toolkit is designed to work with 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.

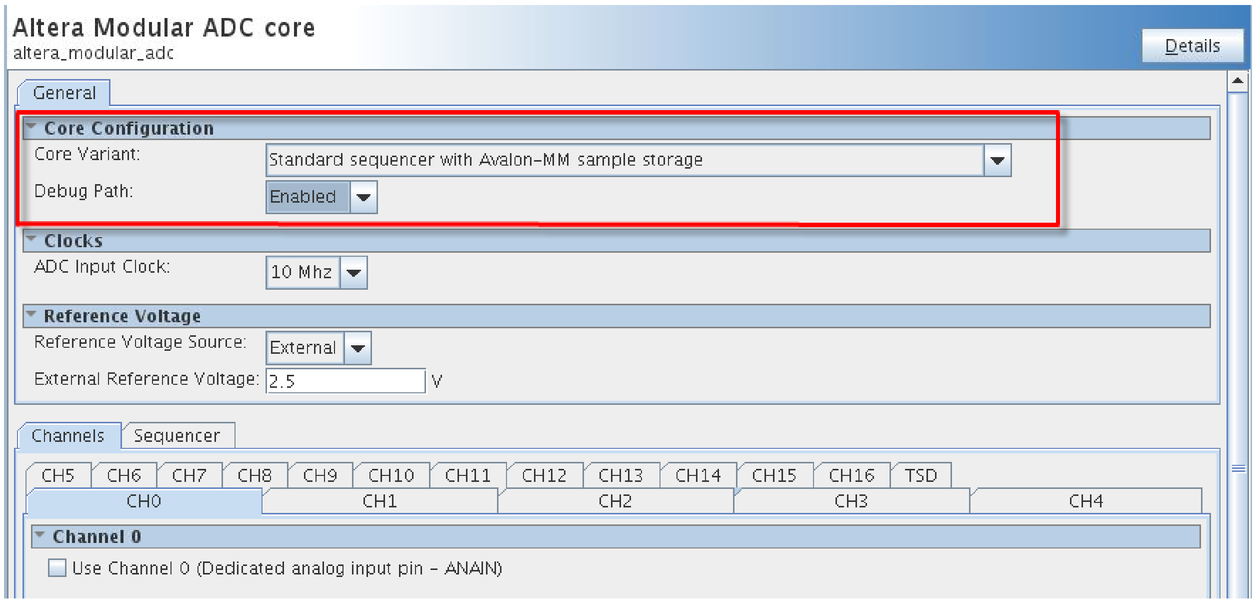
Note: When applying a sine wave, the ADC should sample at 2x the fundamental sine wave frequency. There should be a low-pass filter, 3dB point set to the fundamental frequency.

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 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 6-3: Altera Modular ADC Core



Core Variant: Standard sequencer with Avalon-MM sample storage
Debug Path: Enabled
ADC Input Clock: 10 Mhz
Reference Voltage Source: External
External Reference Voltage: 2.5 V
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.

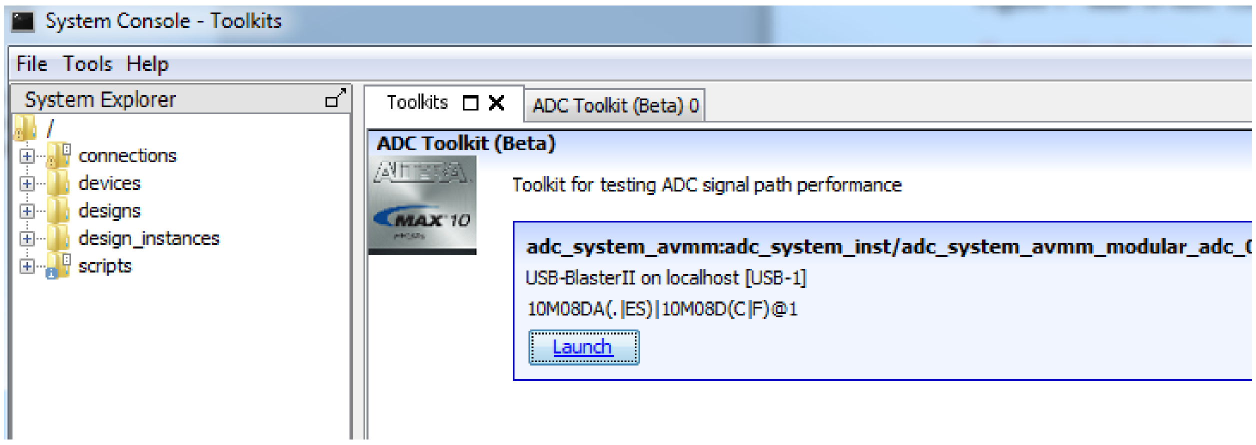
There are two methods to start the ADC Toolkit. Both methods require you to have a MAX 10 device connected, programmed with a project, and linked to this project. However, the **Launch** command only shows up if these requirements are met. You can always start the ADC Toolkit from the **Tools** menu, 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 6-4: 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]

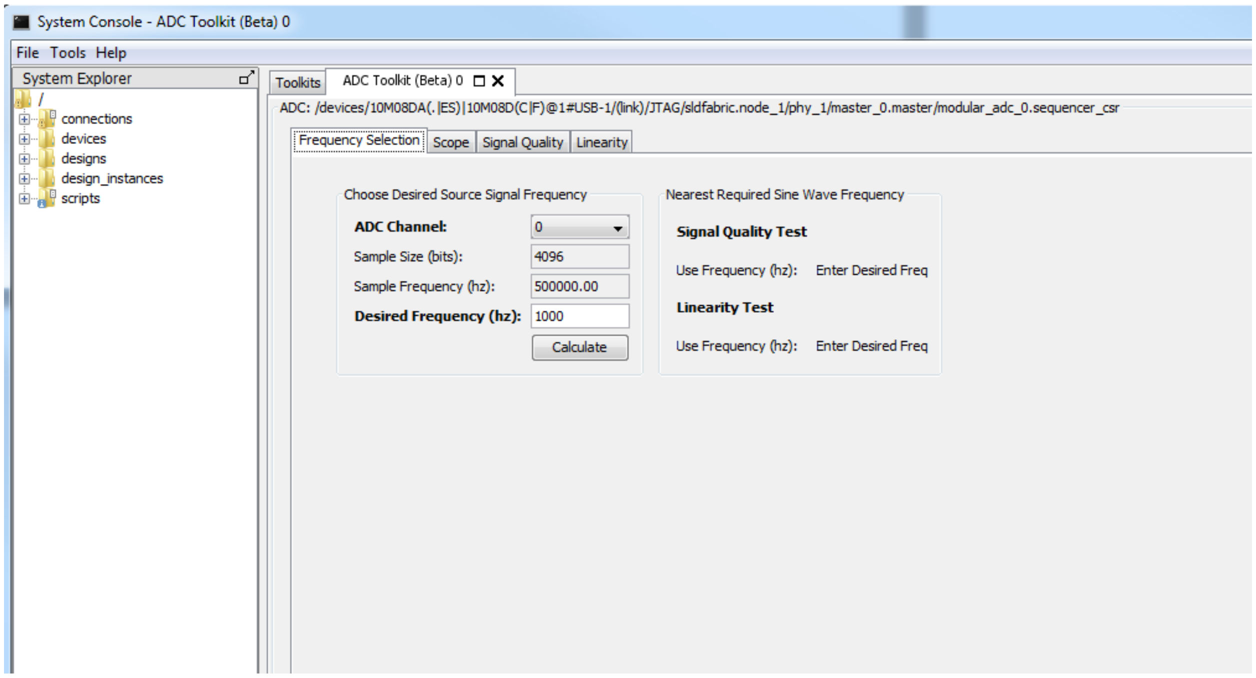
10M08DA(.|ES)|10M08D(C|F)@1

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 6-5: 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 Information

- Using the ADC Toolkit in MAX 10 Devices online training
- MAX 10 FPGA Device Overview
- MAX 10 FPGA Device Datasheet
- MAX 10 FPGA Design Guidelines
- 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.

ADC Toolkit Terms

Table 6-29: ADC Toolkit Terms

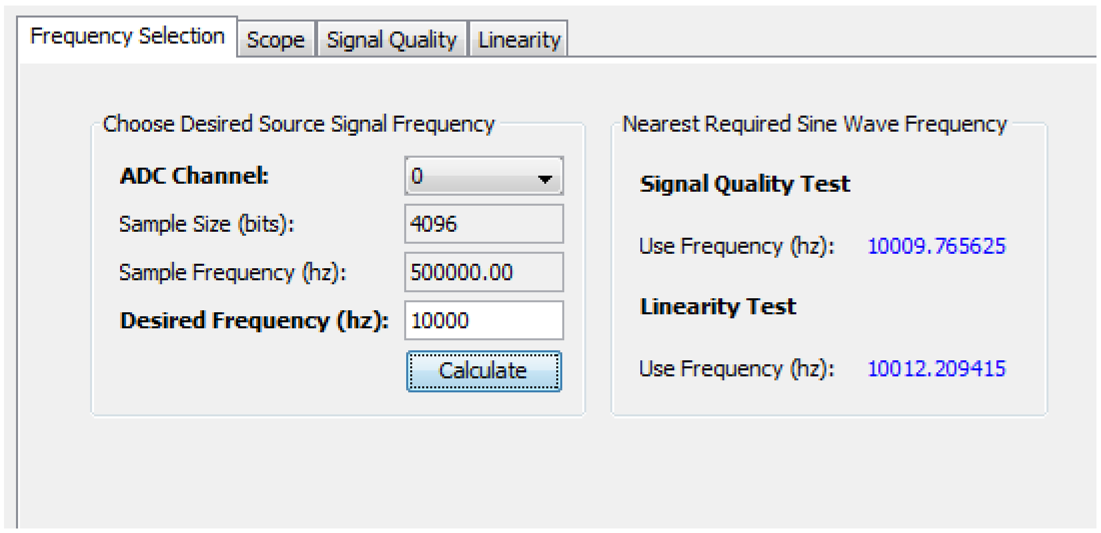
Term	Description
SNR	The ratio of the output signal voltage level to the output noise level.
THD	The ratio of the sum of powers of the harmonic frequency components to the power of the fundamental/original frequency component.
SFDR	Characterizes the ratio between the fundamental signal and the highest spurious in the spectrum.

Term	Description
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.

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 6-6: Frequency Selection Panel


Frequency Selection
Scope
Signal Quality
Linearity
Choose Desired Source Signal Frequency
Nearest Required Sine Wave Frequency
Calculate

ADC Channel:	0
Sample Size (bits):	4096
Sample Frequency (hz):	500000.00
Desired Frequency (hz):	10000

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**.

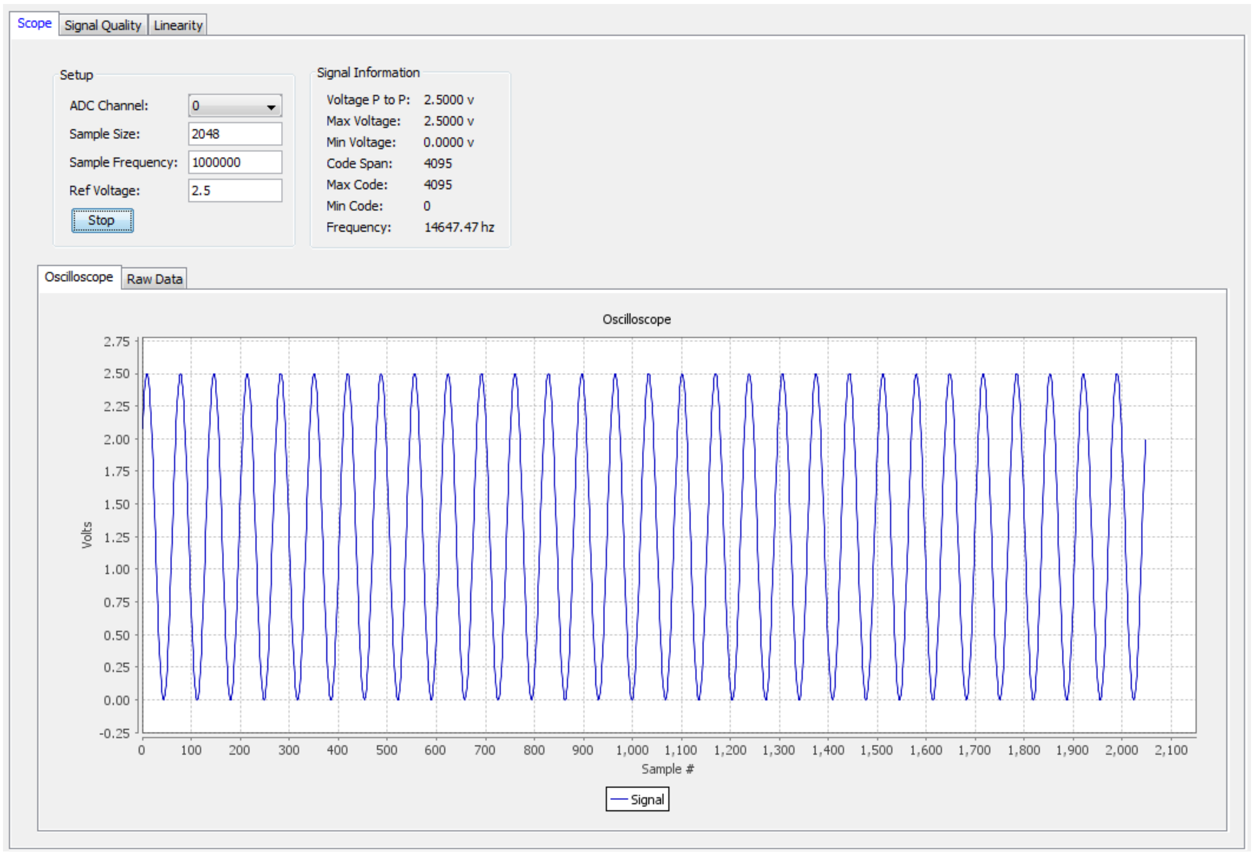
- The closest frequency for valid testing near your desired frequency displays under both **Signal Quality Test** and **Linearity Test**.
- The nearest required sine wave frequencies are different for the signal quality test and linearity test.

4. Set your signal generator to the precise frequency given by the tool based on the type of test you want to run.

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 6-7: Scope Mode Panel



ADC Channel: 0
Sample Size: 2048
Sample Frequency: 1000000
Ref Voltage: 2.5

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 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 Information

Additional information about coherent sampling vs window sampling

Running a Signal Quality Test

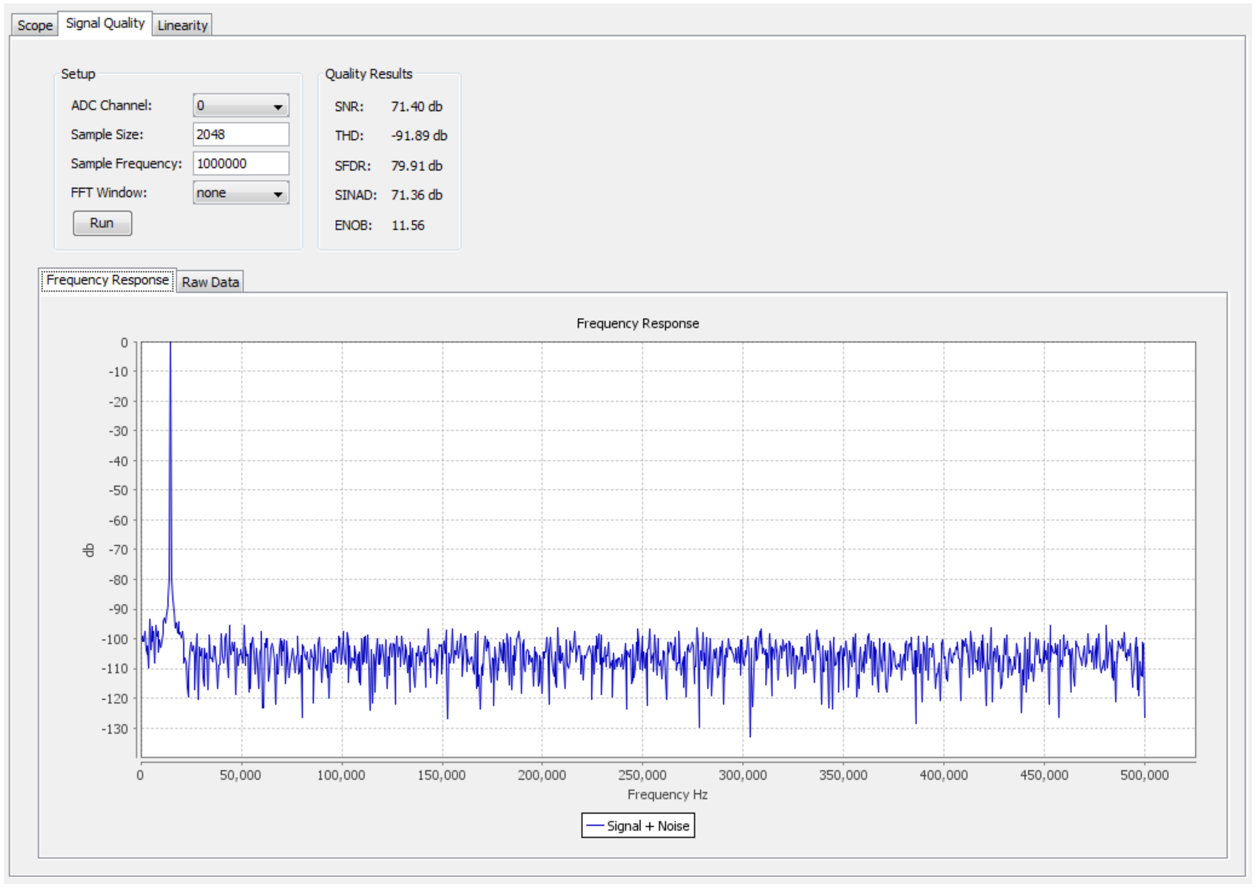
The available performance metrics in signal quality test mode are the following: signal to noise ratio (SNR), total harmonic distortion (THD), spurious free dynamic range (SFDR), signal to noise and distortion ratio (SINAD), effective number of bits (ENOB), and a frequency response graph.
The frequency 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 you begin

Before running a signal quality test, ensure that you have set up the frequency of the reference signal using **Scope** mode.

Figure 6-8: Signal Quality Panel



ADC Channel: 0
Sample Size: 2048
Sample Frequency: 1000000
FFT Window: none

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 Information

Additional information about dynamic parameters such as SNR, THD, etc

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.

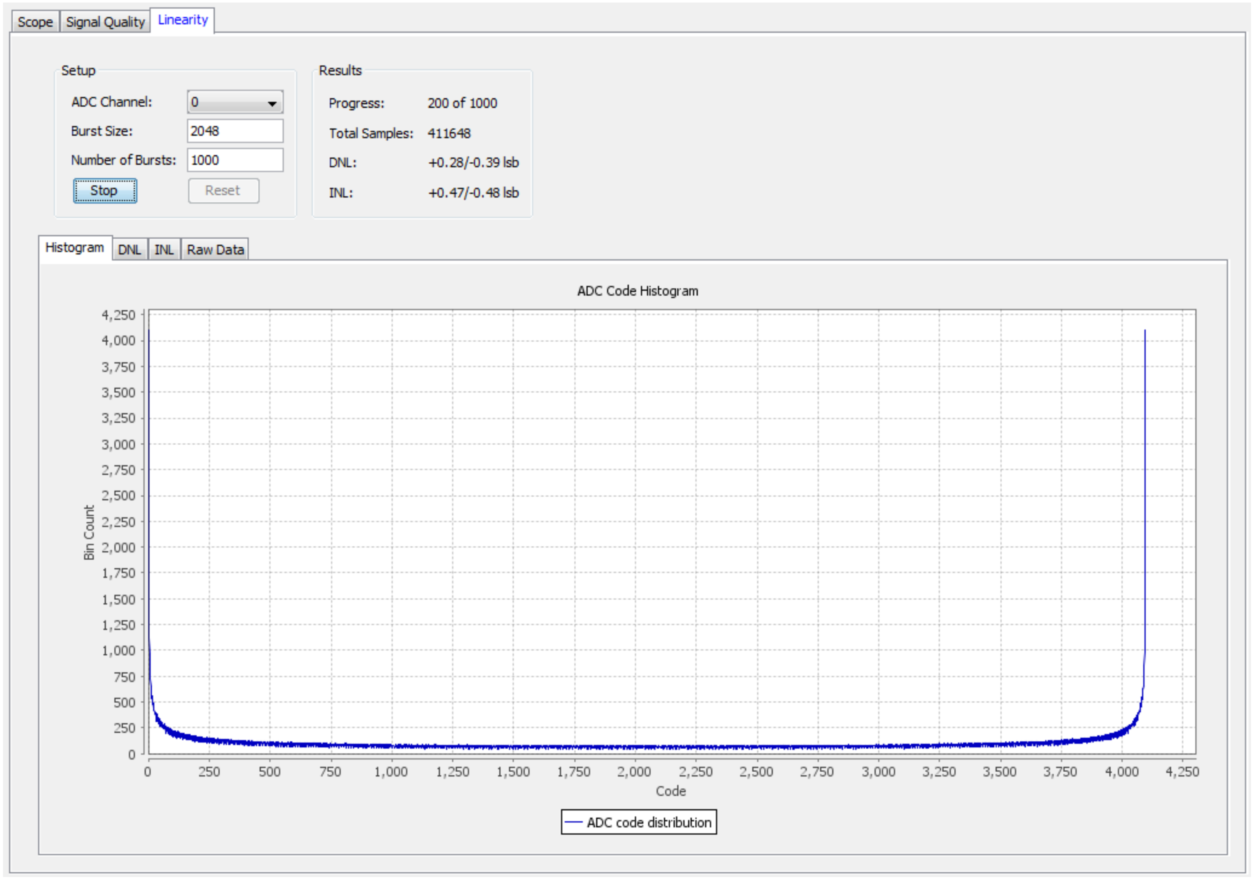
ADC Toolkit Data Views

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 6-9: Example of Pure Sine Wave Histogram

The figure below shows the shape of a pure sine wave signal. Your reference signal should look similar.

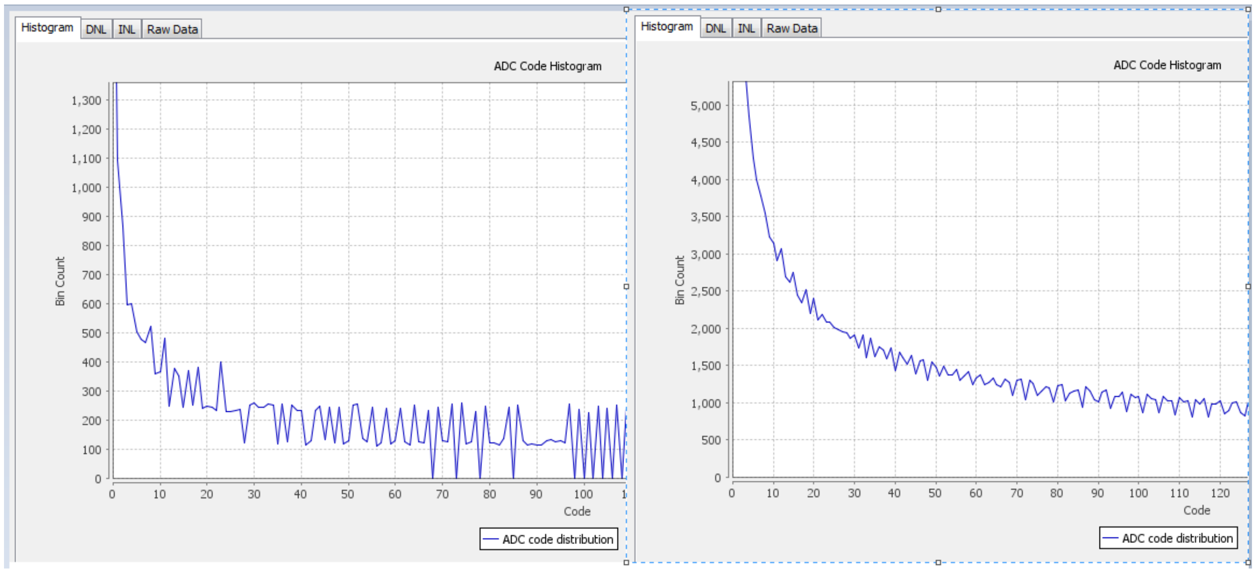


ADC Channel: 0
Burst Size: 2048
Number of Bursts: 1000

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 6-10: Examples of (Left) Poor Frequency Choice vs (Right) Good Frequency Choice

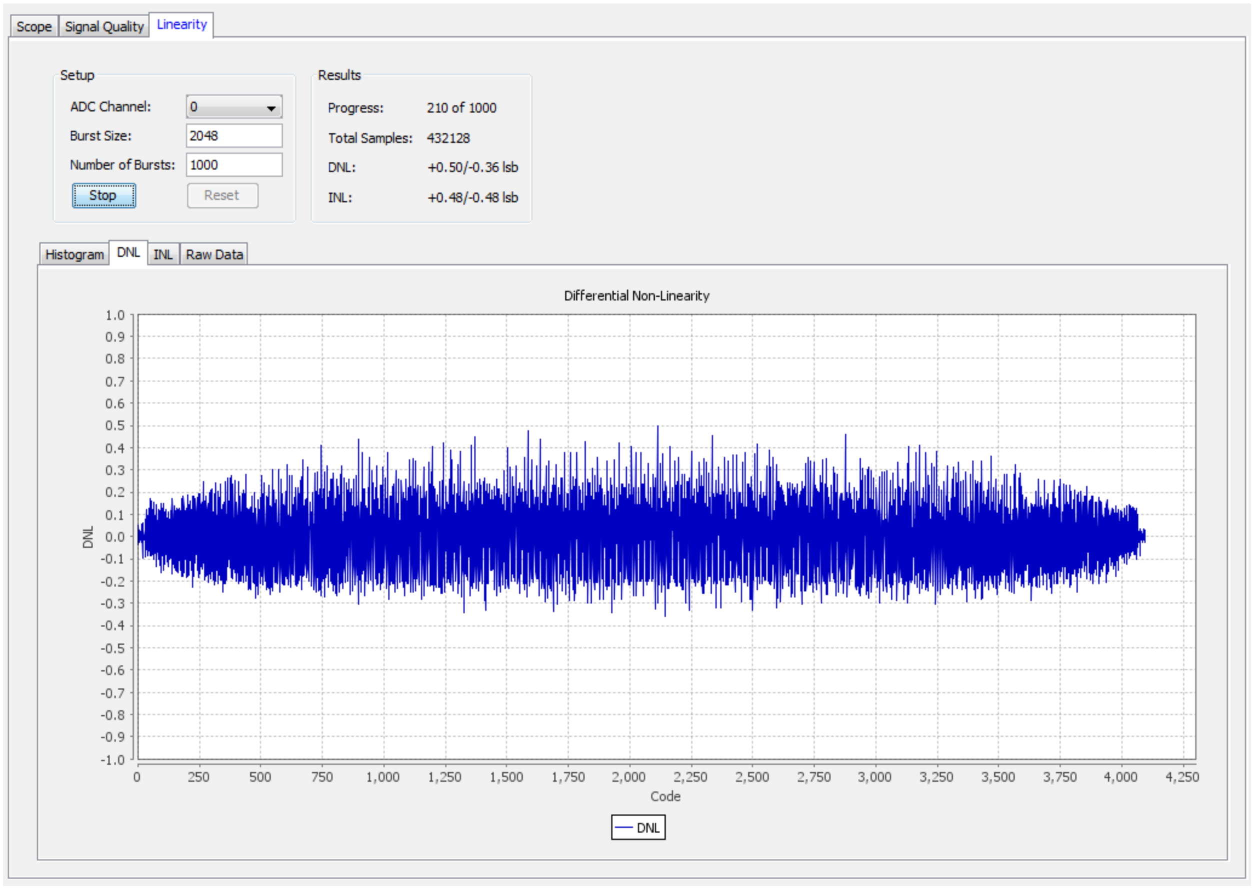


Differential Non-linearity View

Figure 6-11: 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

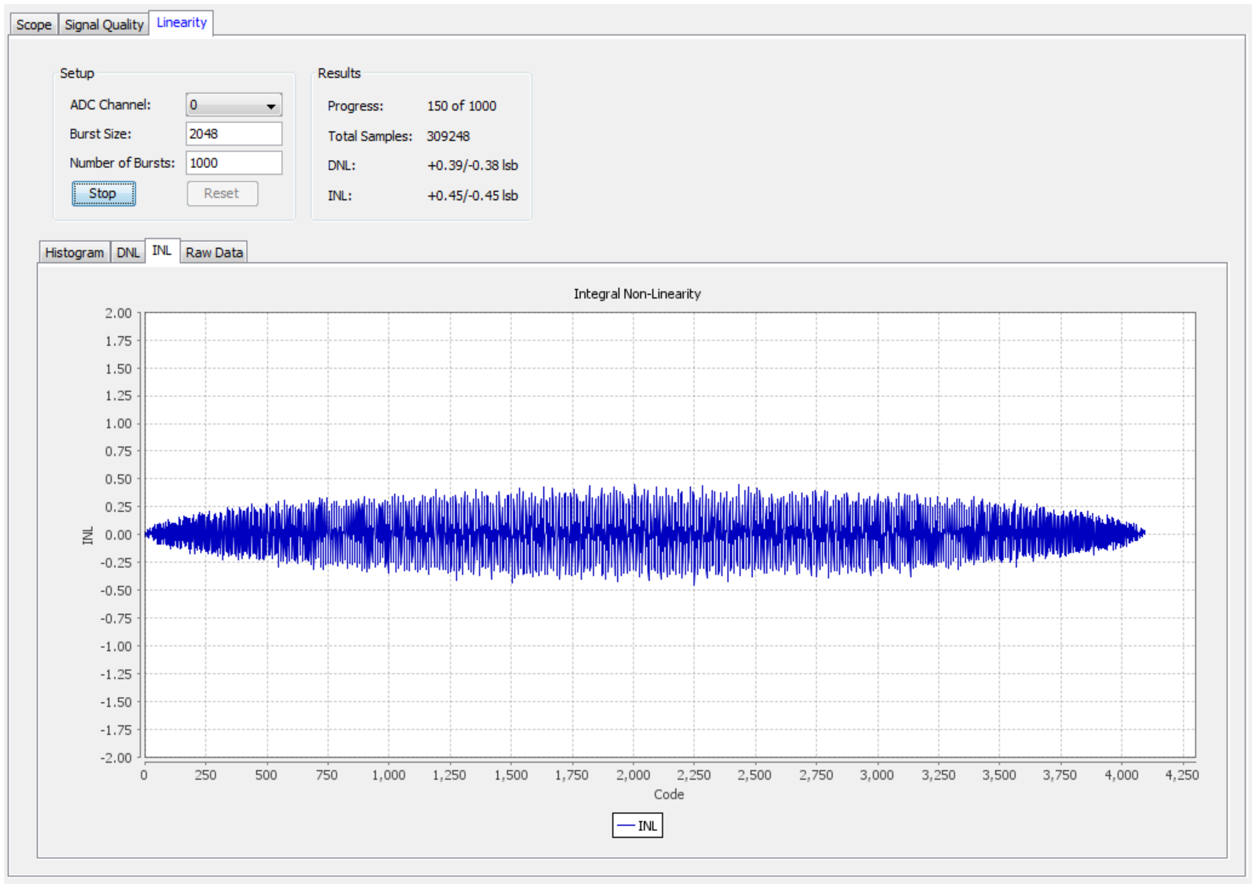
INL: +0.48/-0.48 lsb

Integral Non-linearity View

Figure 6-12: 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.



Setup

ADC Channel:	0
Burst Size:	2048
Number of Bursts:	1000

Results

Progress:	150 of 1000
Total Samples:	309248
DNL:	+0.39/-0.38 lsb
INL:	+0.45/-0.45 lsb

System Console Examples and Tutorials

Altera 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 software, Tcl commands, and Qsys.

Related Information

On-Chip Debugging Design Examples Website

Contains the design files for the example designs that you can download.

Board Bring-Up with System Console Tutorial

You can perform low-level hardware debugging of Qsys systems with System Console. You can debug systems that include IP cores instantiated in your Qsys 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 6-78
   To load the design example into the software and program your device, follow these steps:
2. **Verifying Clock and Reset Signals** on page 6-79
   You can use the System Explorer pane to verify clock and reset signals.
3. **Verifying Memory and Other Peripheral Interfaces** on page 6-79
   The Avalon-MM service accesses memory-mapped slaves via a suitable Avalon-MM master, which can be controlled by the host.
4. **Qsys Modules for Board Bring-up Example** on page 6-85

Related Information

- **Use Cases for System Console**
- **Faster Board Bring-Up with System Console Demo Video**

Setting Up the Board Bring-Up Design Example

To load the design example into the software and program your device, follow these steps:

1. Unzip the **System_Console.zip** file to your local hard drive.
2. Click **File** > **Open Project** and select **Systemconsole_design_example.qpf** with the software.
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 Information

System_Console.zip file
Contains the design files for this tutorial.

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 Information

- **System Explorer Pane** on page 6-7
- **JTAG Debug Commands** on page 6-24

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.

Locating and Opening the Master Service

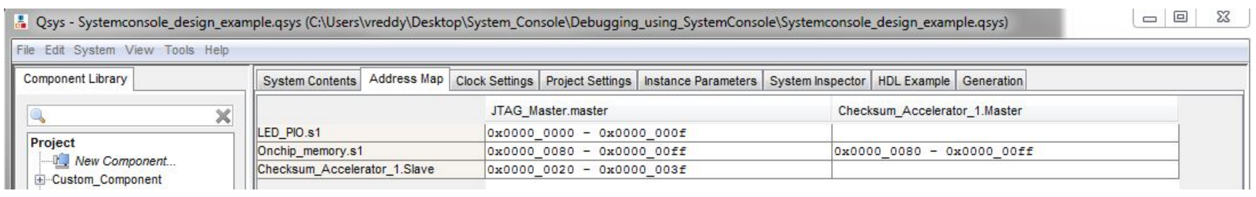
```
#Select the master service type and check for available service paths.
set service_paths [get_service_paths master]

#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]
```

Avalon-MM Slaves

The **Address Map** tab shows the address range for every Qsys component. The Avalon-MM master communicates with slaves using these addresses.

The register maps for all Altera components are in their respective Data Sheets.

Figure 6-13: Address Map


Qsys - Systemconsole_design_example.qsys (C:\Users\vreddy\Desktop\System_Console\Debugging_using_SystemConsole\Systemconsole_design_example.qsys)
File Edit System View Tools Help
Component Library
Project
New Component...
Custom_Component
System Contents
Address Map
Clock Settings
Project Settings
Instance Parameters
System Inspector
HDL Example
Generation

	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 Information

Data Sheets Website

Avalon-MM Commands

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. Master commands can also be used on slave services. If you are working on a slave service, the address field can be a register (if the slave defines register names). [6]

Table 6-30: Avalon-MM Commands

Command	Arguments	Function
`master_write_memory`	*<service-path>* *<address>* *<list_of_byte_values>* *<format>*	Writes the list of byte values. Starts at the specified base address using the most efficient accesses possible with the selected master. The *<format>* argument makes this command accept data as 16 or 32-bit, instead of as bytes. For example: `master_read_memory - format 16 <service_ path> <addr> <count>`
`master_write_8`	*<service-path>* *<address>* *<list_of_byte_values>*	Writes the list of byte values, starting at the specified base address, using 8-bit accesses.
`master_write_16`	*<service-path>* *<address>* *<list_of_16_bit_words>*	Writes the list of 16-bit values, starting at the specified base address, using 16-bit accesses.
`master_write_from_file`	*<service-path>* *<file-name>* *<address>*	Writes the entire contents of the file through the master, starting at the specified 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 list of 32-bit values, starting at the specified base address, using 32-bit accesses.

[6] Transfers performed in 16- and 32-bit sizes are packed in little-endian format.

Command	Arguments	Function
`master_read_memory`	*<service-path>* *<address>* *<size_in_bytes>* *<format>*	Returns a list of <*size*> bytes. Read from memory starts at the specified base address. The <*format*> argument makes this command accept data as 16 or 32-bit, instead of as bytes.
`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.

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 6-31: 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.				

Offset	Register Name	R/W	Fields				
			(n-1)	...	2	1	0
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.
```

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
```

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 6-32: 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		Interrupt 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
```

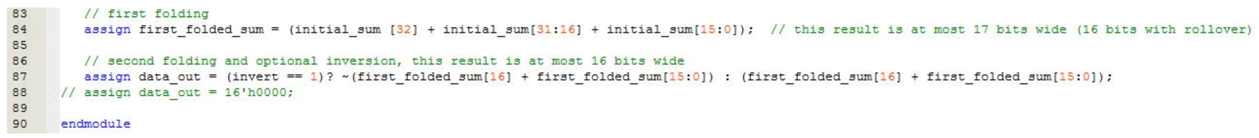
2. If the result is zero and the JTAG chain works properly, the clock and reset signals work properly, and the memory works properly, then the problem is the Checksum Accelerator component.

```
#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 6-14** (uncommented line 87 and comment line 88). Fix the problem.
7. Save the file and regenerate the Qsys 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 6-79 or run the Tcl script included with this design example.

```
source set_memory_and_run_checksum.tcl
```

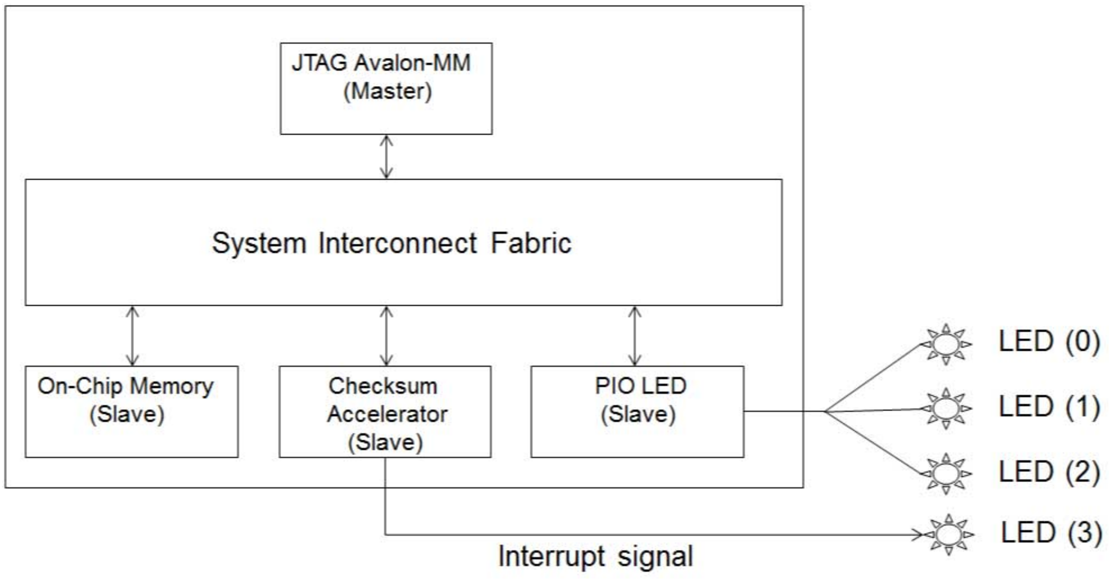
Figure 6-14: Checksum.v File



```
    // first folding
    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)

    // second folding and optional inversion, this result is at most 16 bits wide
    assign data_out = (invert == 1)? ~(first_folded_sum[16] + first_folded_sum[15:0]) : (first_folded_sum[16] + first_folded_sum[15:0]);
 // assign data_out = 16'h0000;

 endmodule
```

Qsys Modules for Board Bring-up Example

Figure 6-15: Qsys Modules for Board Bring-up Example



The Qsys 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**)

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.

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 Information

- Processor Commands on page 6-87
- Nios II Ethernet Standard Design Example
- Nios II Software Build Tools User Guide

Processor Commands

Table 6-33: 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 Information

Nios II Processor Example on page 6-86

[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.

On-Board USB Blaster II Support

System Console supports an On-Board USB-Blaster™ 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 USB-Blaster II. For example, the transceiver signal integrity board does not support the On-Board USB-Blaster II.

About Using MATLAB and Simulink in a System Verification Flow

System Console can be used with MATLAB and Simulink to perform system development testing. You can use the Altera Hardware in the Loop (HIL) tools to set up a system verification flow. In this approach, the design is deployed to hardware and runs in real time. The surrounding components in your system are simulated 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 gives you more 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

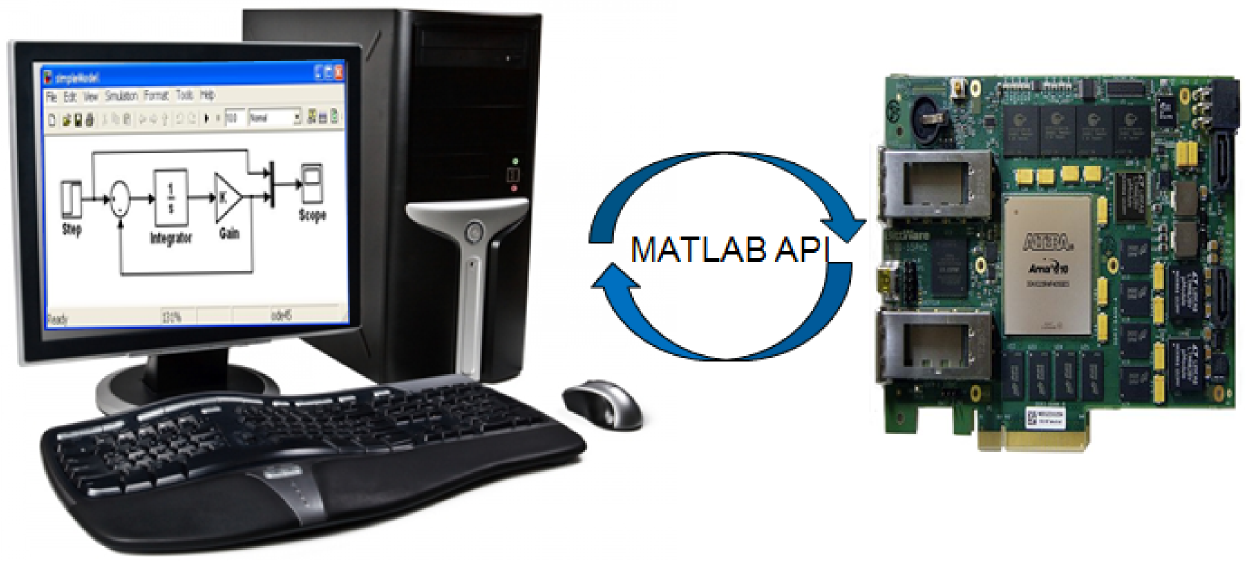
- Avoid long computational delays for algorithms with high processing rates
- API helps to control, debug, visualize, and verify FPGA designs all within the MATLAB environment
- FPGA results are read back by the MATLAB software for further analysis and display

Required Tools and Components

- MATLAB software
- DSP Builder software
- software
- Altera FPGA

Note: The System Console MATLAB API is included in the DSP Builder installation bundle.

Figure 6-16: 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 6-20: 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 software so you have the necessary libraries to enable this flow
2. Build your design using Simulink and the DSP Builder libraries (DSP Builder helps to convert the Simulink design to HDL)
3. Include Avalon-MM components in your design (DSP Builder 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 Qsys
7. Program your Altera FPGA
8. Use the supported MATLAB API commands to interact with your Altera FPGA

Related Information

- **Hardware in the Loop from the MATLAB/Simulink Environment white paper**
- **System in the Loop - Enabling Real-Time FPGA Verification within MATLAB website**
- **DSP Builder website**

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 Quartus Prime software15.1, and later.

Table 6-34: 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.

Document Revision History

Table 6-35: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	• Edits to Toolkit API content and command format. • Added Tookkit API design example. • Added graphic to *Introduction to System Console*. • Deprecated Dashboard. • Changed instances of *Quartus II* to *Quartus Prime*.
2015.11.02	15.1.0	• Edits to Toolkit API content and command format. • Added Tookkit API design example. • Added graphic to *Introduction to System Console*. • Deprecated Dashboard. • Changed instances of *Quartus II* to *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 MATLABS/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.

Date	Version	Changes
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 Information

Quartus Handbook Archive

For previous versions of the *Handbook*, refer to the Quartus Handbook Archive.

Debugging Transceiver Links 7

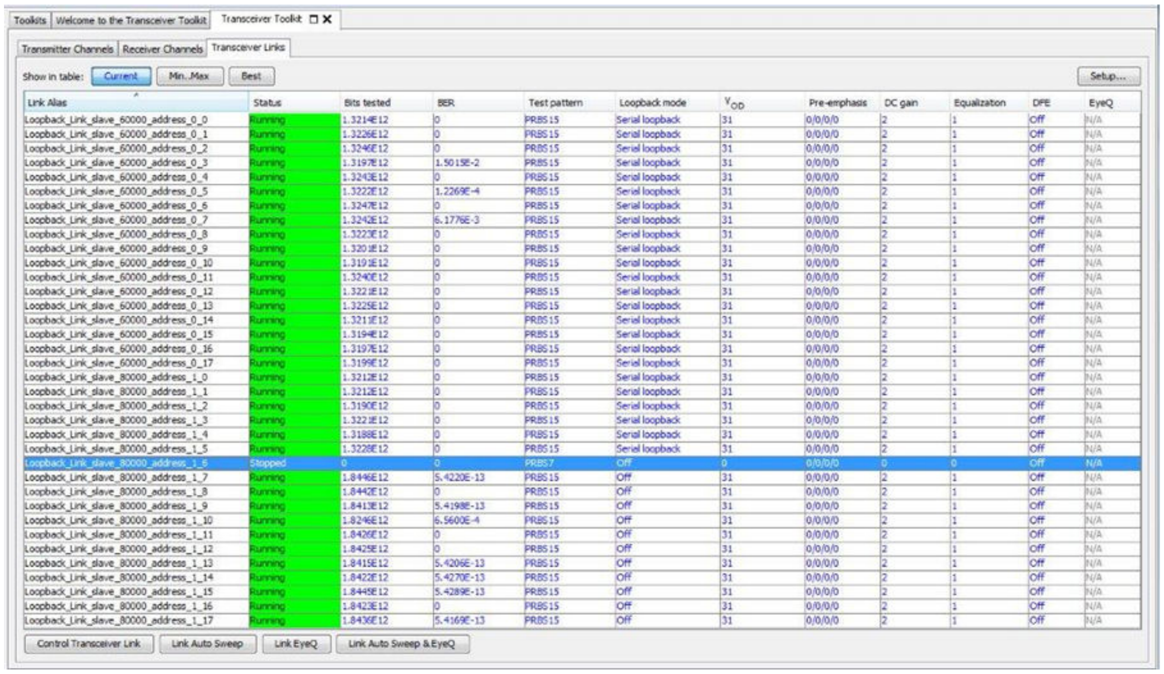
2015.11.02

QPP5V3 Subscribe Send Feedback

The Transceiver Toolkit helps you to optimize high-speed serial links in your board design. The Transceiver Toolkit provides real-time control, monitoring, and debugging of the transceiver links running on your board.

Once you correctly configure a debugging system, you can control the transmitter or receiver channels to optimize transceiver settings and hardware features. The toolkit tests bit-error rate (BER) while running multiple links at target data rate. Run auto sweep tests to identify the best physical media attachment (PMA) settings for each link. EyeQ graphs display the receiver horizontal and vertical eye margin during testing. The toolkit supports testing of multiple devices across multiple boards simultaneously.

Figure 7-1: Transceiver Toolkit Channel Manager



Link Alias	Status	Bits tested	BER	Test pattern	Loopback mode	V_{OD}	Pre-emphasis	DC gain	Equalization	DFE	EyeQ
Loopback_Link_slave_60000_address_0_0	Running	1.3214E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_1	Running	1.3226E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_2	Running	1.3246E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_3	Running	1.3197E12	1.5015E-2	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_4	Running	1.3243E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_5	Running	1.3222E12	1.2269E-4	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_6	Running	1.3247E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_7	Running	1.3242E12	6.1776E-3	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_8	Running	1.3223E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_9	Running	1.3201E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_10	Running	1.3191E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_11	Running	1.3240E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_12	Running	1.3221E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_13	Running	1.3225E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_14	Running	1.3211E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_15	Running	1.3194E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_16	Running	1.3197E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_60000_address_0_17	Running	1.3199E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_0	Running	1.3212E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_1	Running	1.3212E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_2	Running	1.3190E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_3	Running	1.3221E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_4	Running	1.3188E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_5	Running	1.3228E12	0	PRBS15	Serial loopback	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_6	Stopped	0	0	PRBS7	Off	0	0/0/0/0	0	0	Off	N/A
Loopback_Link_slave_80000_address_1_7	Running	1.8446E12	5.4220E-13	PRBS15	Off	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_8	Running	1.8442E12	0	PRBS15	Off	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_9	Running	1.8413E12	5.4198E-13	PRBS15	Off	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_10	Running	1.8246E12	6.5600E-4	PRBS15	Off	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_11	Running	1.8426E12	0	PRBS15	Off	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_12	Running	1.8425E12	0	PRBS15	Off	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_13	Running	1.8415E12	5.4206E-13	PRBS15	Off	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_14	Running	1.8422E12	5.4270E-13	PRBS15	Off	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_15	Running	1.8445E12	5.4289E-13	PRBS15	Off	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_16	Running	1.8423E12	0	PRBS15	Off	31	0/0/0/0	2	1	Off	N/A
Loopback_Link_slave_80000_address_1_17	Running	1.8436E12	5.4169E-13	PRBS15	Off	31	0/0/0/0	2	1	Off	N/A

Transceiver Toolkit User Interface

- **System Explorer**—displays design components and hardware connections.
- **Channel Manager**—control multiple channels simultaneously.
- **Tcl Console**—script control of the Transceiver Toolkit.
- **Messages**—displays information, warning, and error messages.

Quick Start

The Transceiver Toolkit user interface helps you to visualize and debug transceiver links in your design. 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>
```

Get started quickly by downloading Transceiver Toolkit design examples from the **On-Chip Debugging Design Examples** website. For an online demonstration of how to use 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.

Transceiver Debugging Flow

Testing transceiver links involves configuring your system for debug, and then running various link tests.

Table 7-1: Transceiver Link Debugging Flow

	Flow Description
System Configuration Steps	1. Use one of the following methods to define a system that includes necessary transceiver debugging components: • Download Transceiver Toolkit design examples from the **On-Chip Debugging Design Examples** website. Modify Altera design examples to fit your design. • Integrate debugging components into your own design. Click **Tools** > **IP Catalog** to select IP cores and define them in the parameter editor. 2. Click **Assignments** > **Pin Planner** to assign device I/O pins to match your device and board. 3. Click **Tools** > **BluePrint Platform Designer** > to plan a legal device periphery floorplan. 4. Click **Processing** > **Start Compilation** to compile your design. 5. Connect your target device to Altera programming hardware. 6. Click **Tools** > **Programmer** and then program the target device with the debugging system.

	Flow Description
Link Debugging Steps	1. Click **File** > **Load Design**, and select the SRAM Object File (**.sof**) generated for your transceiver design. If you start the toolkit while a project is open, the project loads in the toolkit automatically. 2. (Optional) Create additional links between transmitter and receiver channels. 3. Run any of the following tests: • Run BER with various combinations of PMA settings. • Run PRBS Signal eye tests. • Run custom traffic tests. • Run link optimization tests. • Directly control PMA analog settings to experiment with settings while the link is running.

Related Information

BluePrint Platform Planning

Configuring Systems for Transceiver Debug

To debug transceivers, you must first configure a system that includes the appropriate Altera IP core(s) that support each debugging operation. You can either modify an Altera design example, or configure your own system with required debugging IP components.The debugging system configuration varies by device family. Refer to the appropriate setup information to correctly configure or adapt a system for your target device.

Related Information

Configuring Your Own Debugging System on page 7-4

Configuring an Altera Design Example

Altera provides design examples to help you quickly test your own design. You can experiment with Altera design examples and modify them for your own application. Refer to the **readme.txt** of each design example for more information. Download the Transceiver Toolkit design examples from the On-Chip Debugging Design Examples page of the Altera website.

Use the design examples as a starting point to work with a particular signal integrity development board. The design examples provide the components to quickly test the functionality of the receiver and transmitter channels in your design. 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. You can modify and customize the design examples to match your intended transceiver design.

Once you download the design examples, open the Quartus Prime and click **Project** > **Restore Archived Project** to restore the design example project archive. If you have access to the same development board with the same device as mentioned in the **readme.txt** file of the example, you can directly program the device with the provided programming file in that example. If you want to recompile the design, you must make your modifications to the system configuration in Qsys, regenerate in Qsys, and recompile the design in the Quartus Prime software to generate a new programming file.

If you have the same board as mentioned in the **readme.txt** file, but a different device on your board, you must choose the appropriate device and recompile the design. For example, some early development boards are shipped with engineering sample devices.

You can make changes to the design examples so that you can use a different development board or a different device. If you have a different board, you must edit the necessary pin assignments and recompile the design examples.

Related Information

- **On-Chip Debugging Design Examples**

Configuring Your Own Debugging System

Rather than modifying the Altera Debugging Design Examples, you can integrate debugging IP components into your own design. Refer to the appropriate system configuration steps for your target device.

Related Information

Arria 10 Debug System Configuration on page 7-4

Arria 10 Debug System Configuration

For Arria 10 designs, the Transceiver Toolkit configuration requires instantiation of the Arria 10 Transceiver Native PHY IP core, and use of the built-in Altera Debug Master Endpoint (ADME). Qsys system generation automatically instantiates the JTAG debug link. You enable the ADME features by enabling specific parameters in the Transceiver Native PHY IP core.

Click **Tools** > **IP Catalog** to parameterize, generate, and instantiate the following debugging components.

Table 7-2: Arria 10 / 20nm Transceiver Toolkit IP Core Configuration

Component	Debugging Functions	Parameterization Notes
Transceiver Native PHY	Supports all debugging functions	On the **Dynamic Reconfiguration** tab: • Turn on **Enable dynamic reconfiguration**. • Turn on **Enable odi acceleration logic**. • Turn on **Enable Altera Debug Master Endpoint**. • Turn on **Enable capability registers**. • Turn on **Enable control and status registers**. • Turn on **Enable prbs soft accumulators** (enables hard PRBS Generator and Checker).
Transceiver ATX PLL	Required for Arria 10	On the **Dynamic Reconfiguration** tab: • Turn on **Enable dynamic reconfiguration** (required for System Console read/write access). • Turn on **Enable Altera Debug Master Endpoint** (required for System Console read/write access).
Transceiver PHY Reset Controller	N/A	N/A

Component	Debugging Functions	Parameterization Notes
Altera Debug Master Endpoint (ADME)	• Supports control of PMA analog settings, ADCE settings, DFE settings, and EyeQ. • discovers PHY and use toolkit on designs not using Qsys. • Optionally, turn off to save resource count. (only an option if you instantiate the JTAG to Avalon Master Bridge. Otherwise, Transceiver Toolkit cannot function).	• Enabled from the Arria 10 Transceiver Native PHY Parameter Editor, **Dynamic Reconfiguration** tab.
JTAG Debug Link	Required for Arria 10.	For Qsys projects, the JTAG Debug Link is auto-instantiated.

Enabling Altera Master Debug Endpoint (Arria 10)

The Altera Debug Master Endpoint (ADME) connects to the host link via the system level debug fabric for debugging Arria 10 designs. The ADME provides Avalon-MM Master capability, and is discoverable by System Console. To enable ADME for Arria 10 designs, you must set specific parameters when defining the Arria 10 Transceiver Native PHY and Arria 10 Transceiver ATX PLL IP cores. Click **Tools** > **IP Catalog** to select and define the following IP core parameters. Once properly configured, project synthesis inserts the ADME, debug fabric, and embedded logic.

Figure 7-2: Altera Debug Master Endpoint Block Diagram

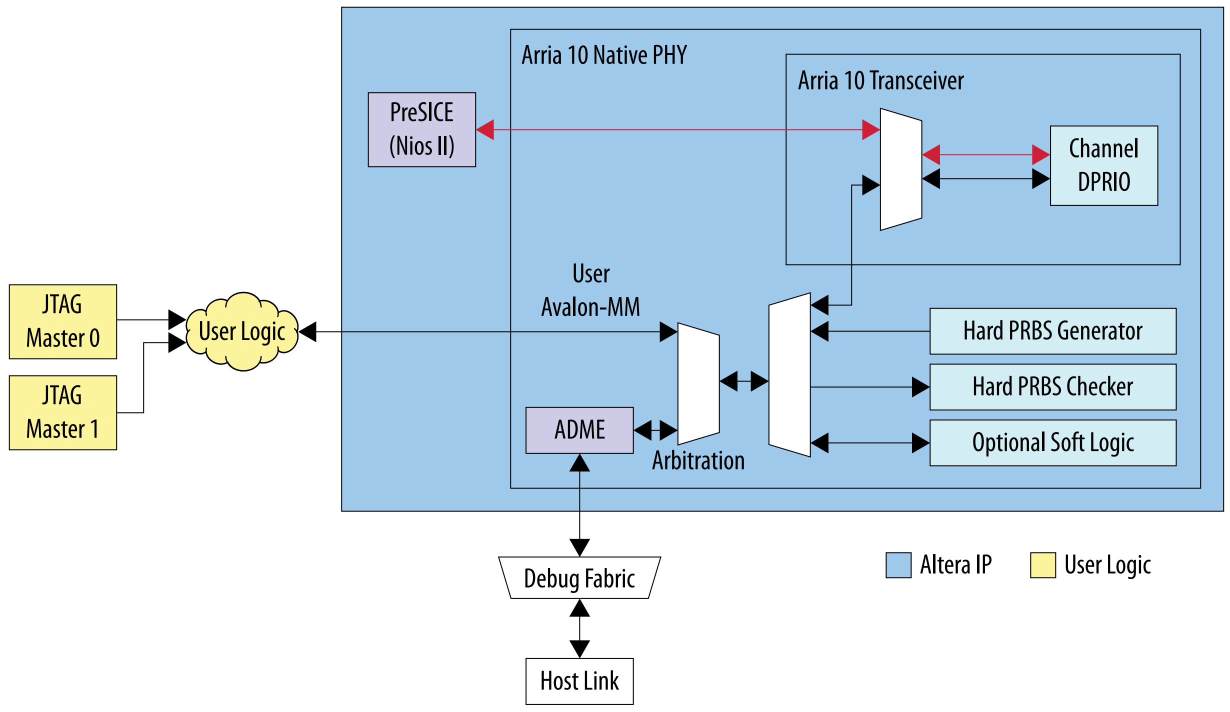
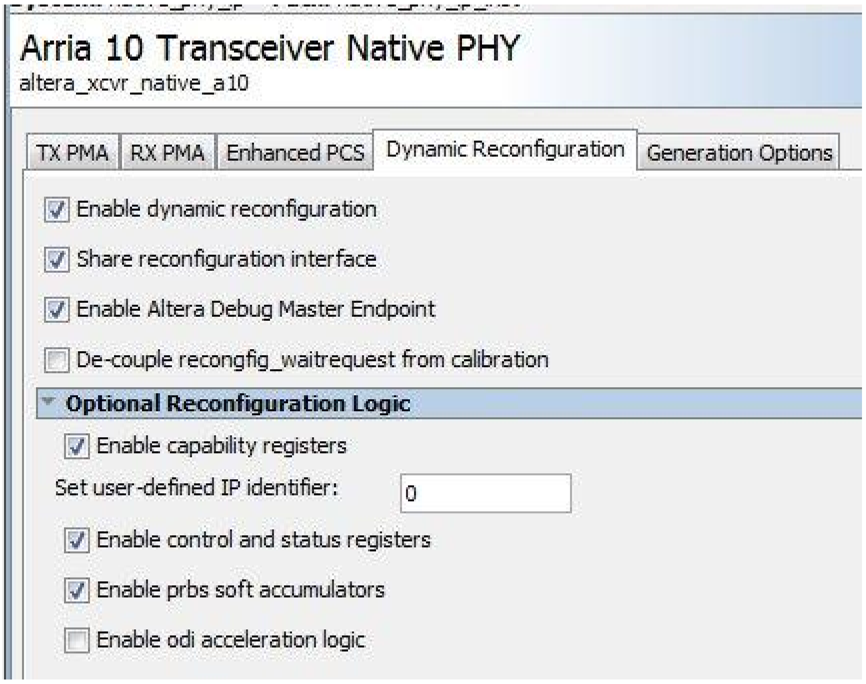


Table 7-3: Enabling ADME IP Core Parameters for Arria 10 Designs

IP Core	Parameter Setting	Location
Arria 10 Transceiver Native PHY	• Turn on **Enable dynamic reconfiguration** • Turn on **Enable Altera Debug Master Endpoint** • Turn on **Enable prbs soft accumulators**	**Dynamic Reconfiguration** tab
Arria 10 Transceiver ATX PLL	• Turn on **Enable capability registers** • Turn on **Enable control and status registers**	**Dynamic Reconfiguration** tab

Figure 7-3: Enabling ADME in A10 Transceiver Native PHY IP Core

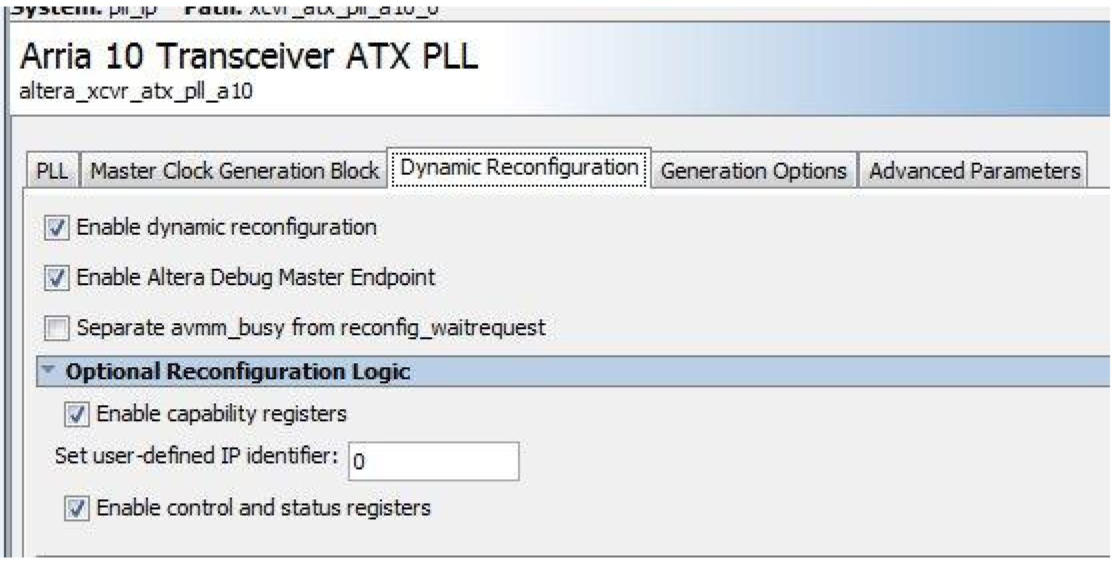

Arria 10 Transceiver Native PHY
altera_xcvr_native_a10

- [✓] Enable dynamic reconfiguration
- [✓] Share reconfiguration interface
- [✓] Enable Altera Debug Master Endpoint
- [ ] De-couple reconfig_waitrequest from calibration

Optional Reconfiguration Logic

- [✓] Enable capability registers
- Set user-defined IP identifier: 0
- [✓] Enable control and status registers
- [✓] Enable prbs soft accumulators
- [ ] Enable odi acceleration logic

Figure 7-4: Enabling ADME in ATX PLL IP Core


Arria 10 Transceiver ATX PLL
altera_xcvr_atx_pll_a10

- [✓] Enable dynamic reconfiguration
- [✓] Enable Altera Debug Master Endpoint
- [ ] Separate avmm_busy from reconfig_waitrequest

Optional Reconfiguration Logic

- [✓] Enable capability registers
- Set user-defined IP identifier: 0
- [✓] Enable control and status registers

Link Testing Configuration (Arria 10)

Use the following system configuration for BER, PRBS, and link optimization testing in Arria 10 devices. To perform BER or PRBS testing of Arria 10 designs, use the PRBS Generator and PRBS Checker functions of the Transceiver Native PHY IP core. You enable the built-in hard PRBS Generator and Checker by turning on **Enable prbs soft accumulators** when configuring the IP core. The Transceiver Toolkit performs required read-write-modify operations on the tested channel(s).

Note: For more information, refer to the *Arria 10 Transceiver Native PHY User Guide*, *Enabling the PRBS and Square Wave Data Generator* and *Enabling the PRBS Data Checker* sections in the *Reconfiguration Interface and Dynamic Configuration* chapter.

The built-in hard PRBS Data Pattern Generator and Checker requires no separate IP instantiation. The hard PRBS Data Pattern Generator and Checker does not support error injection. To use error injection for Arria 10 designs, use the Data Pattern Generator and Checker IP cores in your Arria 10 system design configuration.

Figure 7-5: BER, PRBS, and Link Optimization Test Configuration (Arria 10)

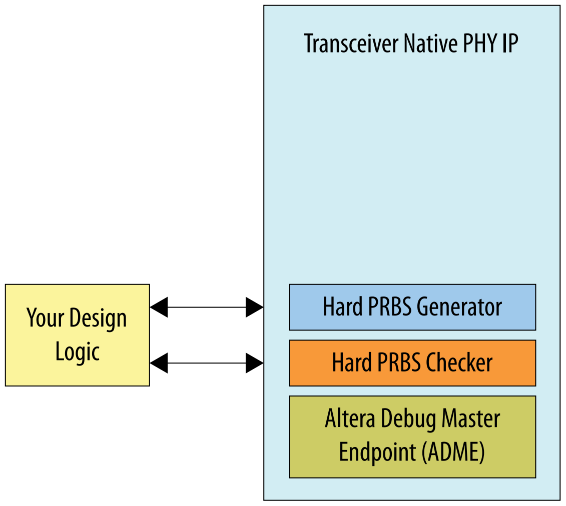

Transceiver Native PHY IP
Your Design Logic
Hard PRBS Generator
Hard PRBS Checker
Altera Debug Master Endpoint (ADME)

Table 7-4: System Connections: BER and PRBS Testing (Arria 10)

From	To
Your design logic	• Transceiver Native PHY Hard PRBS Generator • Transceiver Native PHY Hard PRBS Checker

Custom Traffic Signal Eye Test Configuration (Arria 10)

Use the following configuration for custom traffic signal eye tests in Arria 10 devices.

Figure 7-6: Custom Traffic Signal Eye Test Configuration for Arria 10 / Generation 10 / 20nm

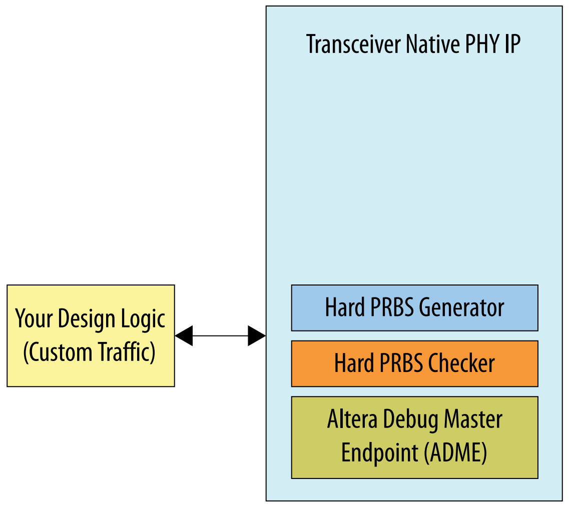

Transceiver Native PHY IP
Your Design Logic (Custom Traffic)
Hard PRBS Generator
Hard PRBS Checker
Altera Debug Master Endpoint (ADME)

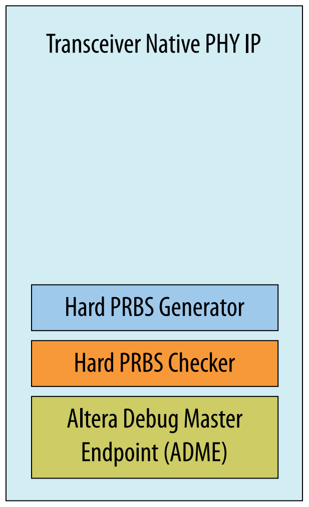
Table 7-5: System Connections: Custom Traffic Signal Eye Test (Arria 10)

From	To
Your design logic custom traffic	• Transceiver Native PHY

PMA Analog Setting Control Configuration (Arria 10)

Use the following configuration to control PMA analog settings in Arria 10 devices.

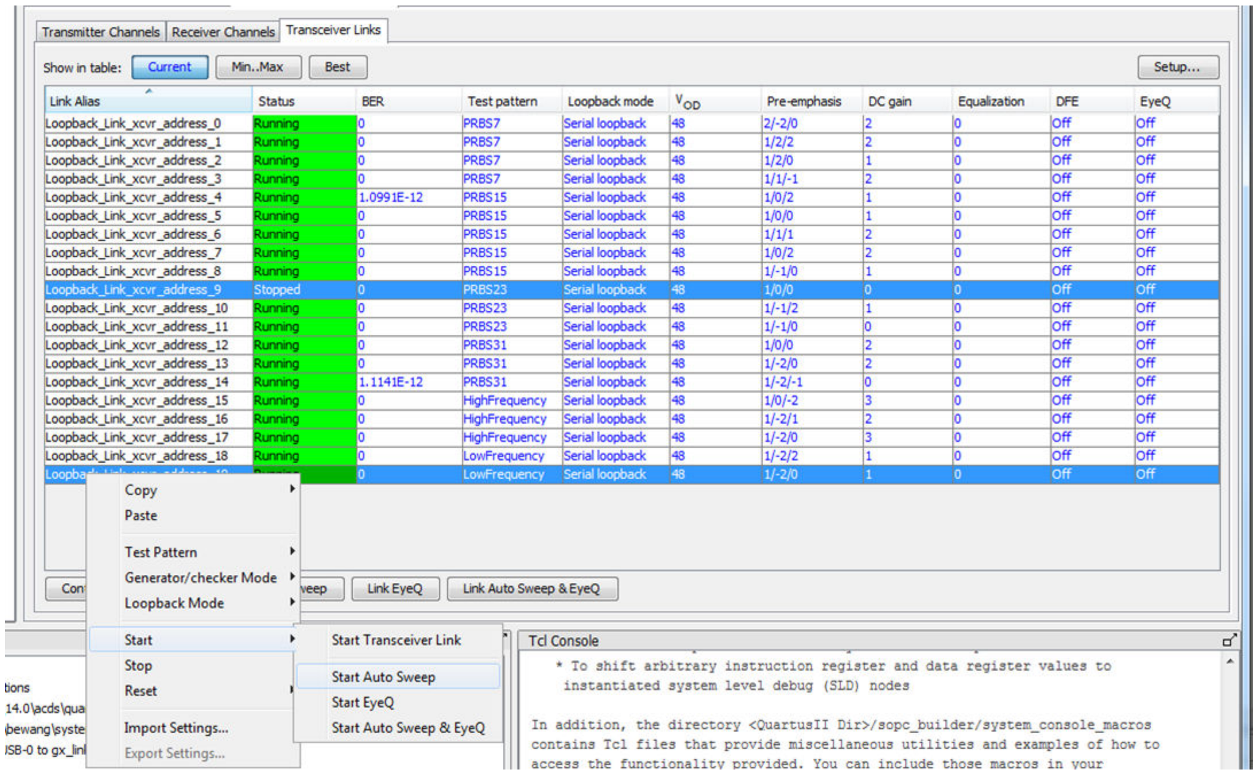
Figure 7-7: System Configuration: PMA Analog Setting Control (Arria 10)


Transceiver Native PHY IP
Hard PRBS Generator
Hard PRBS Checker
Altera Debug Master Endpoint (ADME)

Managing Transceiver Channels

The **Channel Manager** allows you to configure and control large numbers of channels in a spreadsheet view. You can view all the PMA and sweep settings for all channels. You can copy, paste, import, and export settings to and from channels. You can also start and stop sweeps for any or all channels. Right-click in the **Channel Manager** to view additional channel commands. The columns in the **Channel Manager** are movable, resizable, and sortable.

Figure 7-8: Channel Manager GUI



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]	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
```

Copying and Pasting Settings

You can copy PMA and/or sweep settings from a selected row. You can paste PMA and/or sweep settings to one or more rows.

Importing and Exporting Settings

You can select a row in the **Channel Manager** to export your PMA settings to a text file. You can then select one or more rows in the **Channel Manager** to apply the PMA settings from a text file. 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 selected channels.

Starting and Stopping Tests

The **Channel Manager** allows you to start and stop tests by right-clicking the channels. You can select several rows in the **Channel Manager** to start or stop test for multiple channels.

Channel Display Modes

The three display modes are **Current**, **Min/Max**, and **Best**. The default display mode is **Current**.

- **Current**—shows the current values from the device. The blue color 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. Auto sweep cannot be performed on only a transmitter channel; a receiver channel is required to perform an auto sweep test.

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.

Setup Dialog

Click **Setup** from the **Channel Manager** to open the **Setup** dialog box.

Table 7-6: Setup Dialog Popup Menu

Command Name	Action When Clicked	Enabled If
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 row(s) 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.

Controlling Transceiver Channels

You can directly control and monitor transmitters, receivers, and links running on the board in real time. You can transmit a data pattern across the transceiver link, and then report the signal quality of the received data in terms of bit error rate or eye margin with EyeQ.

Click **Control Transmitter Channel** (**Transmitter Channels** tab), **Control Receiver Channel** (**Receiver Channels** tab), or **Control Transceiver Link** (**Transceiver Links** tab) to adjust transmitter or receiver settings while the channels are running.

Debugging Transceiver Links

The Transceiver Toolkit allows you to control and monitor the performance of high-speed serial links running on your board in real-time. You can identify the transceiver links in your design, transmit a data pattern across the transceiver link, and report the signal quality of the received data in terms of bit error rate, bathtub curve, heat map, or EyeQ graph (for supported families).

The toolkit automatically identifies the transceiver links in your design, or you can manually create transceiver links. You can then run auto sweep to help you quickly identify the best PMA settings for each link. You can directly control the transmitter/receiver channels to experiment with various settings suggested by auto sweep. The EyeQ graph allows you to visualize the estimated horizontal and vertical eye opening at the receiver.

The Transceiver Toolkit supports various transceiver link testing configurations. You can identify and test the transceiver link between two Altera devices, or you can transmit a test pattern with a third-party device and monitor the data on an Altera device receiver channel. If a third-party chip includes self-test capability, then you can send the test pattern from the Altera device and monitor the signal integrity at the third-party device receiver channel. If the third-party device supports reverse serial loopback, you can run the test entirely within the Transceiver Toolkit.

Before you can monitor transceiver channels, you must configure a system with debugging components, and program the design into an FPGA. Once those steps are complete, use the following flow to test the channels:

1. Load the design in Transceiver Toolkit
2. Link hardware resources
3. Verify hardware connections
4. Identify transceiver channels
5. Run link tests or control PMA analog settings
6. View results

Step 1: Load Your Design

The Transceiver Toolkit automatically loads the last compiled design upon opening. To load any design into the toolkit, click **File** > **Load Design** and select the **.sof** programming file generated for your transceiver design. Loading the **.sof** automatically links the design to the target hardware in the toolkit. The toolkit automatically discovers links between transmitter and receiver of the same channel. The **System Explorer** displays information about the loaded design.

Step 2: Link Hardware Resources

The toolkit automatically discovers connected hardware and designs. You can also manually link a design to connected hardware resources in the **System Explorer**.

If you are using more than one Altera board, you can set up a test with multiple devices linked to the same design. This setup is useful when you want to perform a link test between a transmitter and receiver on two separate devices. You can also load multiple Quartus Prime projects and make links between different systems. You can perform tests on completely separate and unrelated systems in a single tool instance.

Note: Prior to the Transceiver Toolkit version 11.1, you must manually load and link your design to hardware. In version 11.1 and later, the Transceiver Toolkit automatically links any device programmed with a project.

Figure 7-9: One Channel Loopback Mode for Arria 10 / 20nm

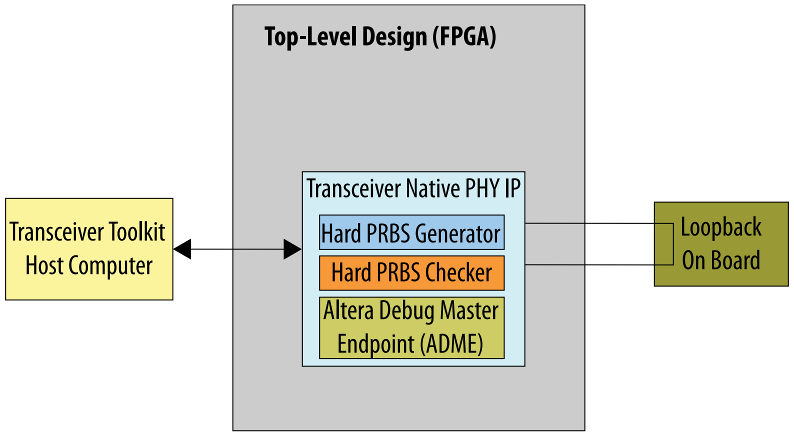
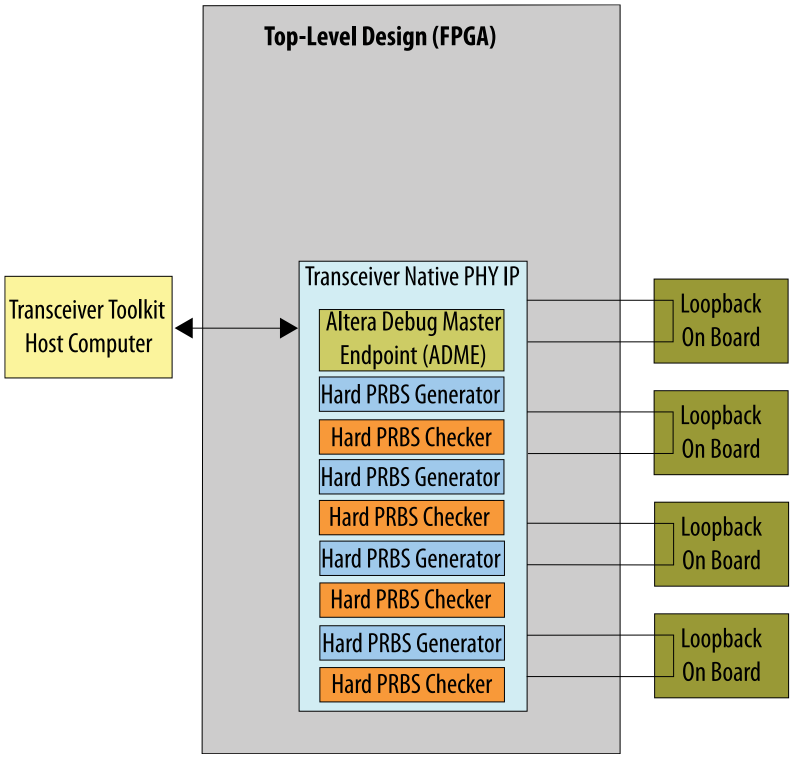

Top-Level Design (FPGA)
Transceiver Toolkit Host Computer
Transceiver Native PHY IP
Hard PRBS Generator
Hard PRBS Checker
Altera Debug Master Endpoint (ADME)
Loopback On Board

Figure 7-10: Four Channel Loopback Mode for Arria 10 / Generation 10 / 20nm



Linking One Design to One Device

To link one design to one device by one USB-Blaster download cable, follow these steps:

1. Load the design for your Quartus Prime project.
2. Link each device to an appropriate design if the design has not auto-linked.
3. Create the link between channels on the device to test.

Linking Two Designs to Two Devices

To link two designs to two separate devices on the same board, connected by one USB-Blaster download cable, follow these steps:

1. Load the design for all the Quartus Prime project files you might need.
2. Link each device to an appropriate design if the design has not auto-linked.
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.

Linking Designs and Devices on Separate Boards

To link two designs to two separate devices on separate boards, connected to separate USB-Blaster download cables, follow these steps:

1. Load the design for all the Quartus Prime project files you might need.
2. Link each device to an appropriate design if the design has not auto-linked.
3. Create the link between channels on the device to test.
4. Link the device you connected to the second USB-Blaster download cable to the second design.
5. Create a link between the channels on the devices you want to test.

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.

Step 3: Verify Hardware Connections

After you load your design and link your hardware, verify that the channels are connected correctly and looped back properly on the hardware. Use the toolkit to send data patterns and receive them correctly. Verifying your link and correct channel before you perform Auto Sweep or EyeQ tests can save time in the work flow.

After you have verified that the transmitter and receiver are communicating with each other, you can create a link between the two transceivers so that you can perform Auto Sweep and EyeQ tests with this pair.

Step 4: Identify Transceiver Channels

The Transceiver Toolkit automatically displays recognized transmitter and receiver channels. The toolkit identifies a channel automatically whenever a receiver and transmitter share a transceiver channel. You can also manually identify the transmitter and receiver in a transceiver channel and create a link between the two for testing.

When you run link tests, channel color highlights indicate the test status:

Table 7-7: 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	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

Step 5: Run Link Tests

Once you identify the transceiver channels for debugging, you can run various link tests in the toolkit.

Use the **Transceiver Links** tab to control link tests. For example, use the Auto Sweep feature to sweep transceiver settings to determine the parameters that support the best BER value. Click **Link Auto Sweep**, **Link EyeQ** or **Link Auto Sweep & EyeQ** to adjust the PMA settings and run tests.

Running BER Tests

You can run BER tests across your transceiver link. After programming the FPGA with your debugging design, loading the design in the toolkit, and linking hardware, follow these steps to run BER 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 **Control Transceiver Link**, and specify a PRBS **Test pattern** and **Data pattern checker** for **Checker mode**. The checker mode option is only available after you turn on **Enable EyeQ block** and **Enable Bit Error Rate Block** in the Reconfiguration Controller component.

   If you select **Bypass** for the **Test pattern**, the toolkit bypasses the PRBS generator and runs your design through the link. The bypass option is only available after you turn on **Enable Bypass interface** in the Reconfiguration Controller component.
3. Experiment with **Reconfiguration**, **Generator**, or **Checker** settings.
4. Click **Start** to run the pattern with your settings. You can then click **Inject Error** to inject error bits, **Reset** the counter, or **Stop** the test.

   Note: Arria 10 devices do not support **Inject Error** if you use the hard PRBS Pattern Generator and Checker in the system configuration.

Related Information

Link Testing Configuration (Arria 10) on page 7-7

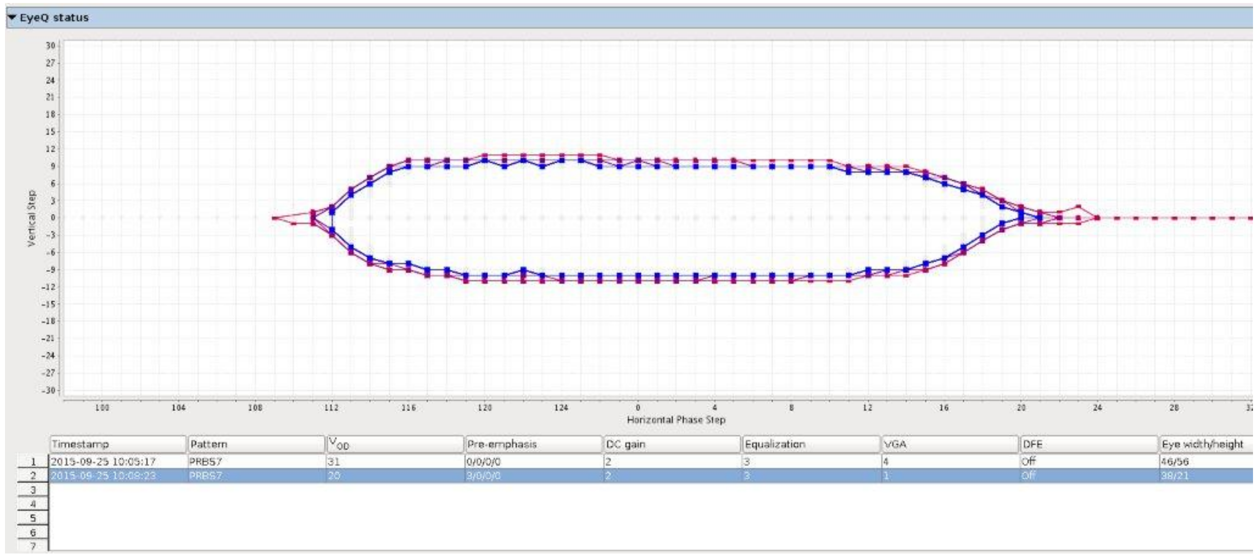
Signal Eye Margin Testing

Some Altera devices include EyeQ 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 EyeQ graph can display a bathtub curve, eye diagram representing eye margin, or heat map display. The run list displays the statistics of each EyeQ 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.

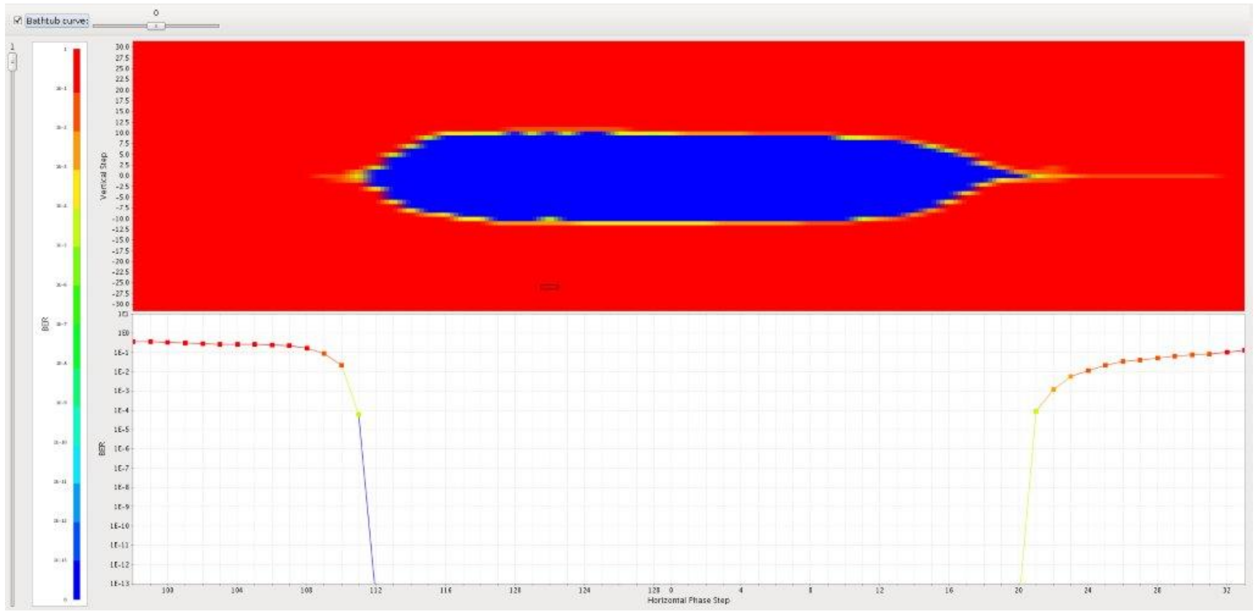
You can right-click any of the test runs in the list, and then click **Apply Settings to Device** to quickly apply those PMA setting to your device. You can also click **Export**, **Import**, or **Create Report**.

Figure 7-11: EyeQ 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 7-12: Heat Map Display and Bathtub Curve Through Eye



Running PRBS Signal Eye Tests

You can 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 EyeQ**, and select **EyeQ** as the **Test mode**. The **EyeQ** mode displays test results as a bathtub curve, heat map, or eye contour representing bit error and phase offset data.
3. Specify **Run length** and **EyeQ settings** to control the test coverage and type of EyeQ results displayed, respectively.
4. Click **Start** to run the pattern with your settings. EyeQ 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 **EyeQ 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.
5. 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 Information

Link Testing Configuration (Arria 10) on page 7-7

Running Custom Traffic Tests

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. Click **Setup** and specify the following:
    a. Select the associated reconfiguration controller.
    b. Click **Create Transceiver Link** and click **Close**.
2. Click the **Receiver EyeQ**, and select **EyeQ** as the **Test mode**. The **EyeQ** 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**.
4. For **Checker mode**, select **Serial bit comparator**.

    The checker mode option is only available after you turn on **Enable EyeQ block** and **Enable Bit Error Rate Block** for the Reconfiguration Controller component.
5. Specify **Run length** and **EyeQ settings** to control the test coverage and type of EyeQ results displayed, respectively.
6. Click **Start** to run the pattern with your settings. EyeQ 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 **EyeQ status**.
7. 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 Information

Custom Traffic Signal Eye Test Configuration (Arria 10) on page 7-9

Auto Sweep Testing

Use the auto sweep feature to automatically sweep ranges for the best transceiver PMA settings. Store a history of the test runs and keep a record of the best PMA settings. Use the best settings the toolkit determines in your final design for improved signal integrity.

Running Link Optimization Tests

After programming the FPGA with your debugging design, loading the design in the toolkit, and linking hardware, follow these steps to run link optimization tests:

1. Click the **Transceiver Links** tab, and 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.
6. When the run completes the chart is displayed and the characteristics of each run are listed in the run list. 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. To use decision feedback equalization (DFE) to determine the best tap settings, follow these steps:
    a. Use Auto Sweep to find optimal PMA settings while leaving the **DFE mode** set to **Off**.
    b. If BER = 0, use the best PMA settings achieved.
    c. 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.
    d. 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 Information

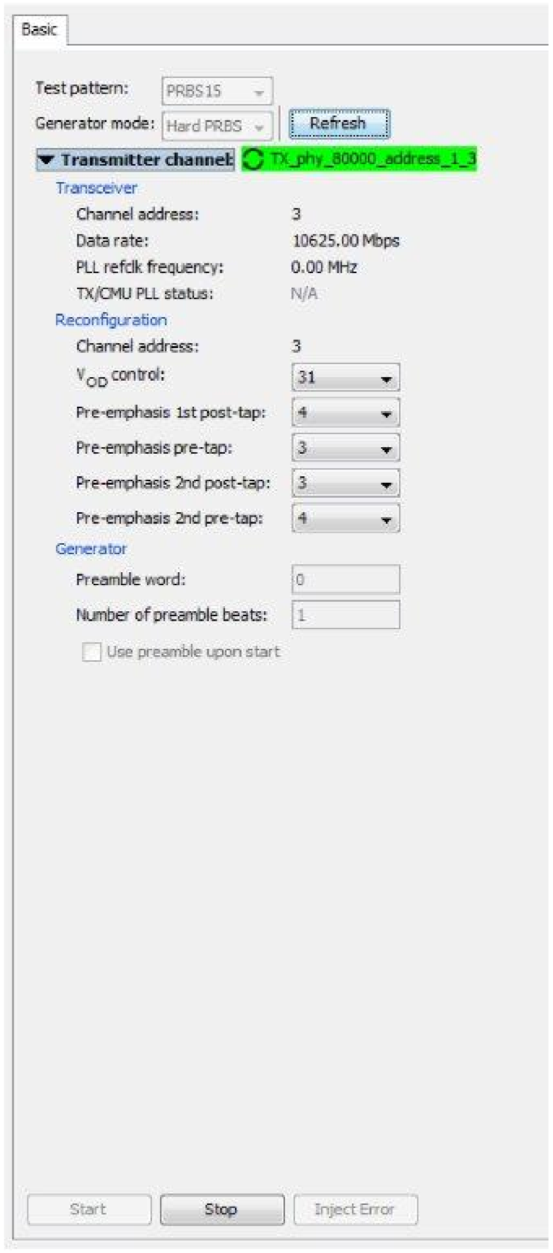
Link Testing Configuration (Arria 10) on page 7-7

Controlling PMA Analog Settings

You can directly control PMA analog settings to experiment with settings while the link is running. To control PMA analog settings, follow these steps:

1. Click **Setup**.
   a. Click the **Transmitter Channels** tab, define a transmitter without a generator, and click **Create Transmitter Channel**.
   b. Click the **Receiver Channels** tab, define a receiver without a generator, and click **Create Receiver Channel**.
   c. Click 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 7-13: 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

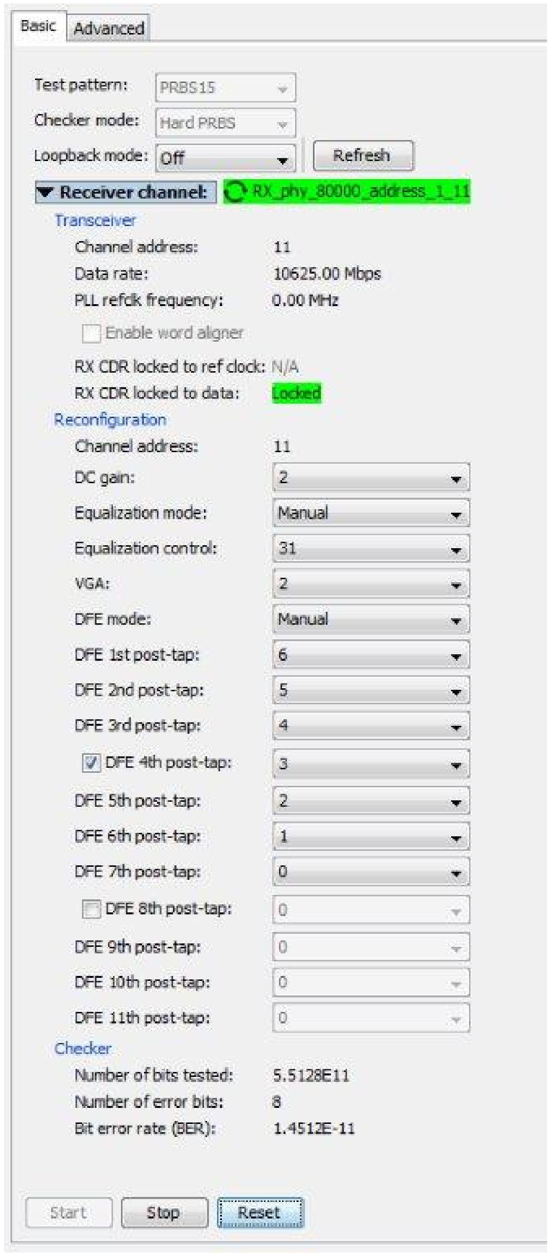
Channel address:	3
V_{OD} control:	31
Pre-emphasis 1st post-tap:	4
Pre-emphasis pre-tap:	3
Pre-emphasis 2nd post-tap:	3
Pre-emphasis 2nd pre-tap:	4

Generator

Preamble word:	0
Number of preamble beats:	1

☐ Use preamble upon start

Figure 7-14: 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

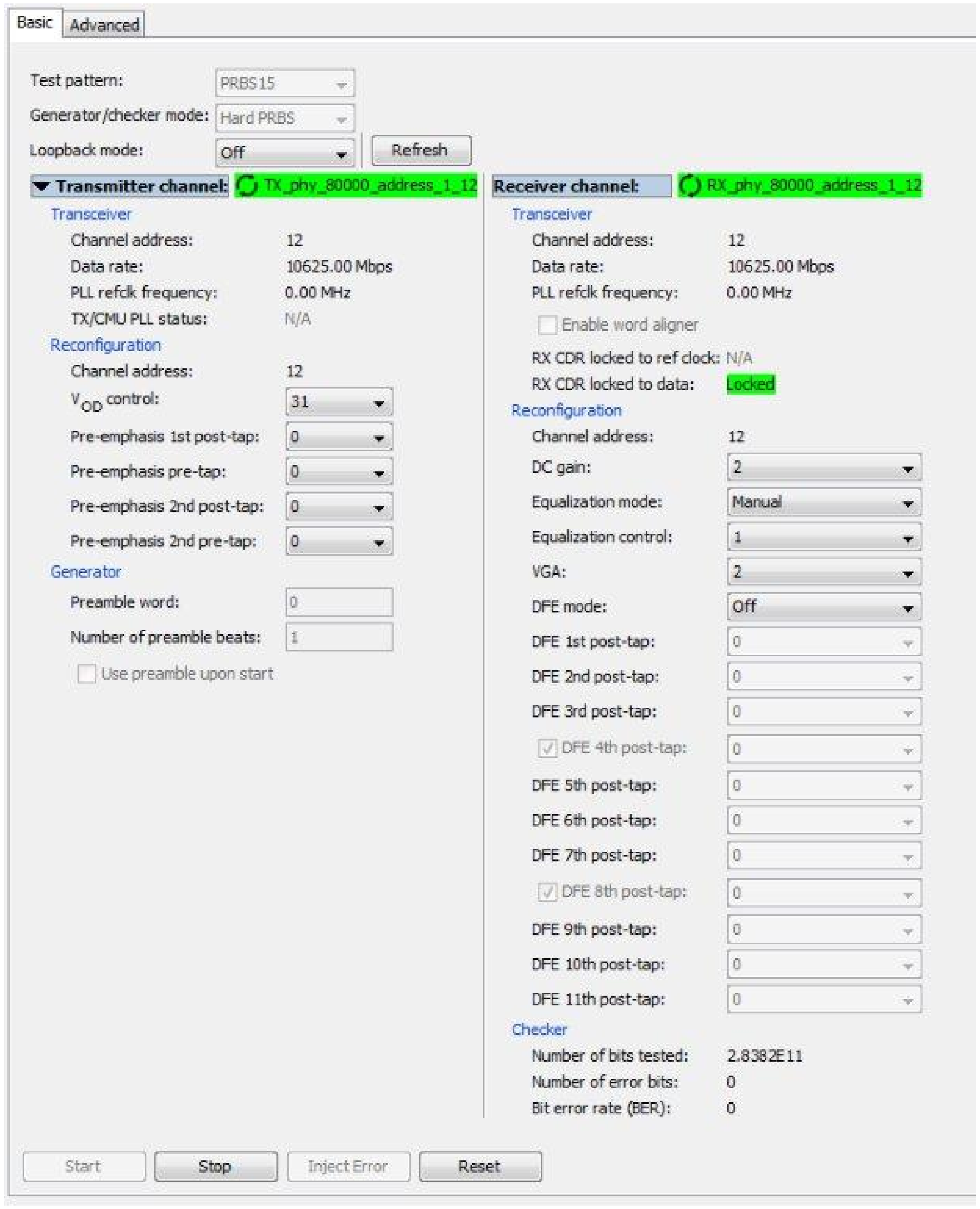
Reconfiguration

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 7-15: 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 Information

PMA Analog Setting Control Configuration (Arria 10) on page 7-9

Troubleshooting Common Errors

- Missing high-speed link pin connections

  The pin connections to identify high-speed links (tx_p/n and rx_p/n) could be missing. When porting an older design to the latest version of the Quartus Prime software, please make sure your connections were preserved.
- 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 which you are trying to set up.
- Unconnected `reconfig_clk`

  The `reconfig_clk` input to the Transceiver Native PHY and ATX PLL IP cores need to be connected and driven. Otherwise, the transceiver link channel will not be displayed.

User Interface Settings Reference

The following settings are available for interaction with the transmitter channels or receiver channels or transceiver links in the Transceiver Toolkit user interface.

Table 7-8: Transceiver Toolkit Control Panel Settings

Setting	Description	Control Panel
Alias	Name you choose for the channel.	Transmitter Receiver Transceiver Link
Auto sweep status	Reports the current and best tested bits, errors, bit error rate, and case count for the current auto sweep test.	Receiver Transceiver Link
Bit error rate (BER)	Specifies errors divided by bits tested since the last reset of the checker.	Receiver Transceiver Link
Channel address	Logical address number of the transceiver channel.	Transmitter Receiver Transceiver Link

Setting	Description	Control Panel
Checker mode	Specify **Data pattern checker** or **Serial bit comparator** for BER tests. If you enable **Serial bit comparator** the Data Pattern Generator sends the PRBS pattern, but the pattern is checked by the serial bit comparator. In **Bypass mode**, clicking **Start** begins counting on the Serial bit comparator. For BER testing: • Arria 10 devices supports the Data Pattern Checker and the Hard PRBS. For EyeQ testing: • Arria 10 devices Arria 10 support the Serial Bit Checker. • Stratix V devices support the Data Pattern Checker and the Serial Bit Checker.	Receiver Transceiver Link
Data rate	Data rate of the channel as read from the project file or data rate as measured by the frequency detector. To use the frequency detector, turn on **Enable Frequency Counter** in the Data Pattern Checker IP core and/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 as read from the project file. Click the refresh button next to the measured **Data rate** if you make changes to your settings and want to sample the data rate again.	Transmitter Receiver Transceiver Link
DC gain	Circuitry that provides an equal boost to the incoming signal across the frequency spectrum.	Receiver Transceiver Link
DFE mode • Values 1-11 (Arria 10 devices)	Decision feedback equalization (DFE) for improving signal quality. DFE modes are **Off**, **Manual** and **Continuous Adaptation** in Arria 10 devices. Continuous mode DFE automatically tries to find the best tap values.	Receiver Transceiver Link

Setting	Description	Control Panel
Equalization control	Boosts the high-frequency gain of the incoming signal, thereby compensating for the low-pass filter effects of the physical medium. When used with DFE, use DFE in **Manual** or **Continuous** mode.	Receiver Transceiver Link
Equalization mode	You can set **Equalization Mode** to **Manual** or **Triggered** for Arria 10 devices.	Receiver Transceiver Link
Error rate limit	Turns on or off error rate limits. **Start checking after** waits until the set number of bits are satisfied until it starts 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 Transceiver Link
EyeQ mode	Allows you to specify Eye contour or Bathtub curve as the type of EyeQ graph generated by the test.	Transmitter Receiver Transceiver Link
EyeQ phase step	Sets the phase step for sampling the data from an offset of the CDR (clock data recovery) data path; set to Off to use the regular clock data recovery (CDR) data path.	Receiver Transceiver Link
EyeQ status	Displays a graphical representation of signal integrity as an eye contour, bathtub curve plot, or heat map.	Transmitter Receiver Transceiver Link
EyeQ vertical step	Sets the voltage threshold of the sampler to report the height of the eye. Negative numbers are allowed for vertical steps to capture asymmetric eye.	Receiver Transceiver Link
Horizontal phase step interval	Specify 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 Receiver Transceiver Link

Setting	Description	Control Panel
Increase test range	Right-click in the **Advanced** panel to use the span capabilities of Auto Sweep to automatically increase the span of tests by one unit down for the minimum and one unit up for the maximum, for the selected set of controls. 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 could yield better results.	Receiver Transceiver Link
Maximum tested bits	Sets the maximum number of tested bits for each test iteration.	Receiver Transceiver Link
Number of bits tested	Specifies the number of bits tested since the last reset of the checker.	Receiver Transceiver Link
Number of error bits	Specifies the number of error bits encountered since the last reset of the checker.	Receiver Transceiver Link
PLL refclk freq	Channel reference clock frequency as read from the project file or measured reference clock frequency as calculated from the measured data rate.	Transmitter Receiver Transceiver Link
Populate with	Right-click in the **Advanced** panel to load current values on the device as a starting point, or initially load the best settings determined through auto sweep. The Quartus Prime software automatically applies the values you specify in the drop-down lists for the Transmitter settings and Receiver settings.	Receiver Transceiver Link
Preamble word	Word to send out if you use the preamble mode (only if you use soft PRBS Data Pattern Generator and Checker).	Transmitter Transceiver Link
Pre-emphasis	The programmable pre-emphasis module in each transmit buffer boosts high frequencies in the transmit data signal, which may be attenuated in the transmission media. Using pre-emphasis can maximize the data eye opening at the far-end receiver.	Transmitter Transceiver Link
Receiver channel	Specifies the name of the selected receiver channel.	Receiver Transceiver Link

Setting	Description	Control Panel
Refresh Button	After loading the **.pof**, loads fresh settings from the registers after running dynamic reconfiguration.	Transmitter Receiver Transceiver Link
Reset	Resets the current test.	Receiver Transceiver Link
Rules Based Configuration (RBC) validity checking	Displays invalid combination of settings in red in each list under **Transmitter settings** and **Receiver settings**, based on previous settings. If selected, the settings remain in red to indicate the currently selected combination is invalid. This avoids manually testing invalid settings that you cannot compile for your design. This prevents setting the device into an invalid mode for extended periods of time and potentially damaging the circuits.	Receiver Transceiver Link
Run length	Sets coverage parameters for test runs.	Transmitter Receiver Transceiver Link
Run results table	Lists the statistics of each EyeQ test run. The run results table is sortable. You can right-click any of the tests in the table and then click **Apply Settings to Device** to quickly apply the chosen PMA settings to your device. You can click **Import** to load reports from previously generated EyeQ runs into the run results table. You can click **Export** to export single or multiple runs from the run results table to a report.	Transmitter Receiver Transceiver Link
RX CDR PLL status	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 Transceiver Link
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 toggles.	Receiver Transceiver Link

Setting	Description	Control Panel
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 Receiver Transceiver Link
Start	Starts the pattern generator or checker on the channel to verify incoming data.	Transmitter Receiver Transceiver Link
Stop	Stops generating patterns and testing the channel.	Transmitter Receiver Transceiver Link
Target bit error rate	Finds the contour edge of the bit error rate that you select. This option only applies to eye contour mode.	Transmitter Receiver Transceiver Link
Test mode	Allows you to specify the **Auto sweep**, **EyeQ**, or **Auto sweep and EyeQ** test mode.	Receiver Transceiver Link
Test pattern	Test pattern sent by the transmitter channel. Arria 10 devices support **PRBS9**, **PRBS15**, **PRBS23**, and **PRBS31**).	Transmitter Receiver Transceiver Link
Time limit	Specifies the time limit unit and value to have a maximum bounds time limit for each test iteration	Receiver Transceiver Link
Transmitter channel	Specifies the name of the selected transmitter channel.	Transmitter Transceiver Link
TX/CMU PLL status	Provides status of whether the transmitter channel PLL is locked to the reference clock.	Transmitter Transceiver Link
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 Transceiver Link

Setting	Description	Control Panel
Vertical phase step interval	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 Receiver Transceiver Link
VGA	The variable gain amplifier (VGA) amplifies the signal amplitude and ensures a constant voltage swing before the data is fed to the CDR for sampling. This assignment controls the VGA output voltage swing and can be set to any value from 0 to 7. **Note:** Supports Arria 10 devices only	Receiver Transceiver Link
V_{OD} control	Programmable transmitter differential output voltage.	Transmitter Transceiver Link

Scripting API Reference

You can alternatively use Tcl commands to access Transceiver Toolkit functions, rather than using the GUI. You can script various tasks, such as loading a project, creating design instances, linking device resources, and identifying high-speed serial links. You can save your project setup in a Tcl script for use in subsequent testing sessions. You can also build a custom test routine script.

After you set up and define links that describe the entire physical system, you can click **Save Tcl Script** to save the setup for future use. To run the scripts, double-click script names in the System Explorer scripts folder.

View a list of the available Tcl commands in the Tcl Console window. Select Tcl commands in the list to view descriptions, including example usage.

To view Tcl command descriptions from the Tcl Console window:

1. Type `help help`. The Console displays all Transceiver Toolkit Tcl commands.
2. Type `help` <*command name*>. The Console displays the command description.

Transceiver Toolkit Commands

The following tables list the available Transceiver Toolkit scripting commands.

Table 7-9: 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.

Command	Arguments	Function
`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 specified channel at the specified tap position.
`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>*	Gets whether this channel has the DFE feature available.
`transceiver_channel_rx_has_eyeq`	*<service-path>*	Gets whether the EyeQ feature is available for the specified channel.
`transceiver_channel_rx_is_checking`	*<service-path>*	Returns non-zero if the checker is running.
`transceiver_channel_rx_is_dfe_enabled`	*<service-path>*	Gets whether the DFE feature is enabled on the specified channel.
`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 specified channel.
`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 specified channel.
`transceiver_channel_rx_set_dfe_tap_value`	*<service-path> <tap position> <tap value>*	Sets the current tap value of the specified channel at the specified tap position to the specified value.
`transceiver_channel_rx_set_dfe_adaptive`	*<service-path>*	Sets the mode of DFE adaptation. 0=off, 1=adaptive, 2= one-time adaptive

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_eyeq_phase_step`	*<service-path>*	Gets the current phase step of the specified channel.
`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_eyeq_enabled`	*<service-path>*	Gets whether the EyeQ feature is enabled on the specified channel.
`transceiver_channel_rx_set_eyeq_enabled`	*<service-path> <disable(0)/enable(1)>*	Enables or disables the EyeQ feature on the specified channel.
`transceiver_channel_rx_set_eyeq_phase_step`	*<service-path><phase step>*	Sets the phase step of the specified channel.
`transceiver_channel_rx_set_word_aligner_enabled`	*<service-path><disable(0)/enable(1)>*	Enables or disables the word aligner of the specified channel.
`transceiver_channel_rx_is_word_aligner_enabled`	*<service-path><disable(0)/enable(1)>*	Gets whether the word aligner feature is enabled on the specified channel.
`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_eyeq_1d`	*<service-path>*	Detects whether the eye viewer pointed to by *<service-path>* supports 1D-EyeQ mode.
`transceiver_channel_rx_set_1deye_mode`	*<service-path><disable(0)/enable(1)>*	Enables or disables 1D-EyeQ mode.
`transceiver_channel_rx_get_1deye_mode`	*<service-path>*	Returns the current on or off status of 1D-EyeQ mode.

Table 7-10: 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 currently set number of beats to send out the preamble word.
`transceiver_channel_tx_get_pattern`	*<service-path>*	Returns the currently set pattern.
`transceiver_channel_tx_get_preamble_word`	*<service-path>*	Returns the currently set preamble word.
`transceiver_channel_tx_get_preemph0t`	*<service-path>*	Gets the pre-emphasis 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_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.

Command	Arguments	Function
`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><preemph0t value>*	Sets the pre-emphasis pre-tap value on the transmitter channel.
`transceiver_channel_tx_set_preemph1t`	*<service-path><preemph1t value>*	Sets the pre-emphasis first post-tap value on the transmitter channel.
`transceiver_channel_tx_set_preemph2t`	*<service-path><preemph2t value>*	Sets the pre-emphasis second post-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 7-11: Transceiver Toolkit Transceiver Toolkit Debug_Link Commands

Command	Arguments	Function
`transceiver_debug_link_get_pattern`	*<service-path>*	Gets the currently set 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 7-12: Transceiver Toolkit Reconfig_Analog Commands

Command	Arguments	Function
`transceiver_reconfig_analog_get_logical_channel_address`	*<service-path>*	Gets the transceiver logical 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 logical channel address.
`transceiver_reconfig_analog_get_rx_eqctrl`	*<service-path>*	Gets the equalization control value on the receiver channel specified by the current logical channel address.
`transceiver_reconfig_analog_get_tx_preemph0t`	*<service-path>*	Gets the pre-emphasis pre-tap value on the transmitter channel specified by the current logical 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 logical 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 logical channel address.
`transceiver_reconfig_analog_get_tx_vodctrl`	*<service-path>*	Gets the V_{OD} control value on the transmitter channel specified by the current logical channel address.
`transceiver_reconfig_analog_set_logical_channel_address`	*<service-path><logical channel address>*	Sets the transceiver logical 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 logical 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 logical channel address.

Command	Arguments	Function
`transceiver_reconfig_analog_set_tx_preemph0t`	*<service-path><preemph0t value>*	Sets the pre-emphasis pre-tap value on the transmitter channel specified by the current logical channel address.
`transceiver_reconfig_analog_set_tx_preemph1t`	*<service-path><preemph1t value>*	Sets the pre-emphasis first post-tap value on the transmitter channel specified by the current logical channel address.
`transceiver_reconfig_analog_set_tx_preemph2t`	*<service-path> <preemph2t value>*	Sets the pre-emphasis second post-tap value on the transmitter channel specified by the current logical 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 logical channel address.

.

Table 7-13: Transceiver Toolkit Decision Feedback Equalization (DFE) Commands

Command	Arguments	Function
`alt_xcvr_reconfig_dfe_get_logical_channel_address`	*<service-path>*	Gets the logical channel address that other `alt_xcvr_reconfig_dfe` commands use to apply.
`alt_xcvr_reconfig_dfe_is_enabled`	*<service-path>*	Gets whether the DFE feature is enabled on the previously specified channel.
`alt_xcvr_reconfig_dfe_set_enabled`	*<service-path> <disable(0)/enable(1)>*	Enables or disables the DFE feature on the previously specified channel.
`alt_xcvr_reconfig_dfe_set_logical_channel_address`	*<service-path> <logical channel address>*	Sets the logical 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 specified channel at specified tap position and value.

Table 7-14: Transceiver Toolkit Eye Monitor Commands

Command	Arguments	Function
`alt_xcvr_custom_is_word_aligner_enabled`	*<service-path> <disable(0)/enable(1)>*	Gets whether the word aligner feature is enabled on the previously specified channel.
`alt_xcvr_custom_set_word_aligner_enabled`	*<service-path> <disable(0)/enable(1)>*	Enables or disables the word aligner of the previously specified channel.
`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 specified channel is enabled.
`alt_xcvr_custom_set_serial_loopback_enabled`	*<service-path> <disable(0)/enable(1)>*	Enables or disables the serial loopback mode of the previously specified channel.
`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 logical channel address on which other `alt_reconfig_eye_viewer` commands will use to apply.
`alt_xcvr_reconfig_eye_viewer_get_phase_step`	*<service-path>*	Gets the current phase step of the previously specified channel.
`alt_xcvr_reconfig_eye_viewer_is_enabled`	*<service-path>*	Gets whether the EyeQ feature is enabled on the previously specified channel.

Command	Arguments	Function
`alt_xcvr_reconfig_eye_viewer_set_enabled`	*<service-path> <disable(0)/enable(1)>*	Enables or disables the EyeQ feature on the previously specified channel. Setting a value of 2 enables both EyeQ and the Serial Bit Comparator.
`alt_xcvr_reconfig_eye_viewer_set_logical_channel_address`	*<service-path> <logical channel address>*	Sets the logical channel address on which other `alt_reconfig_eye_viewer` commands will use to apply.
`alt_xcvr_reconfig_eye_viewer_set_phase_step`	*<service-path> <phase step>*	Sets the phase step of the previously specified channel.
`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>*	Gets 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.
`alt_xcvr_reconfig_eye_viewer_has_1deye`	*<service-path>*	Detects whether the eye viewer pointed to by *<service-path>* supports 1D-EyeQ mode.

Command	Arguments	Function
`alt_xcvr_reconfig_eye_viewer_set_1deye_mode`	*<service-path> <disable(0)/enable(1)*	Enables or disables 1D-EyeQ mode.
`alt_xcvr_reconfig_eye_viewer_get_1deye_mode`	*<service-path>*	Gets the enable or disabled state of 1D-EyeQ mode.

Table 7-15: 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 7-16: 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.

Command	Arguments	Function
`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.

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 7-17: 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.

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.

Command	Arguments	Function
`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 7-18: 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.
`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.

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 7-19: 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.

Command	Arguments	Function
`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 7-20: 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.

Document Revision History

Table 7-21: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *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.
2015.11.02	15.1.0	• 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.

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 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**.

Date	Version	Changes
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, 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.
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 Information

Quartus Handbook Archive

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.

Quick Design Debugging Using SignalProbe

8

2015.11.02

QPP5V3 Subscribe Send Feedback

Quick Design Debugging Using SignalProbe

The SignalProbe 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 SignalProbe 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 SignalProbe feature supports the Arria® series, Cyclone® series, MAX® II, and Stratix® series device families.

Related Information

- **System Debugging Tools Overview documentation** on page 5-1
  Overview and comparison of all the tools available in the Quartus Prime software

Design Flow Using SignalProbe

The SignalProbe feature allows you to reserve available pins and route internal signals to those reserved pins, while preserving the behavior of your design. SignalProbe is an effective debugging tool that provides visibility into your FPGA.

You can reserve pins for SignalProbe and assign I/O standards after a full compilation. Each SignalProbe-source to SignalProbe-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 SignalProbe, perform the following tasks:

1. Perform a full compilation.
2. Reserve SignalProbe Pins.
3. Assign SignalProbe sources.
4. Add registers between pipeline paths and Signalprobe pins.
5. Perform a SignalProbe compilation.
6. Analyze the results of a SignalProbe compilation.

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**.

Reserve SignalProbe Pins

SignalProbe 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 SignalProbe pins. The sources for SignalProbe pins are the internal nodes and registers in the post-compilation netlist that you want to probe.

Note: Although you can reserve SignalProbe pins using many features within the Quartus Prime software, including the Pin Planner and the Tcl interface, you should use the **SignalProbe Pins** dialog box to create and edit your SignalProbe pins.

Assign SignalProbe Sources

A SignalProbe source can be any combinational node, register, or pin in your post-compilation netlist. To find a SignalProbe source, in the Node Finder, use the SignalProbe 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 SignalProbe 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.

Because SignalProbe pins are implemented and routed as ECOs, turning the **SignalProbe 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 Information

- **SignalProbe Pins Dialog Box online help**
- **Add SignalProbe Pins Dialog Box online help**
- **Engineering Change Management with the Chip Planner documentation**

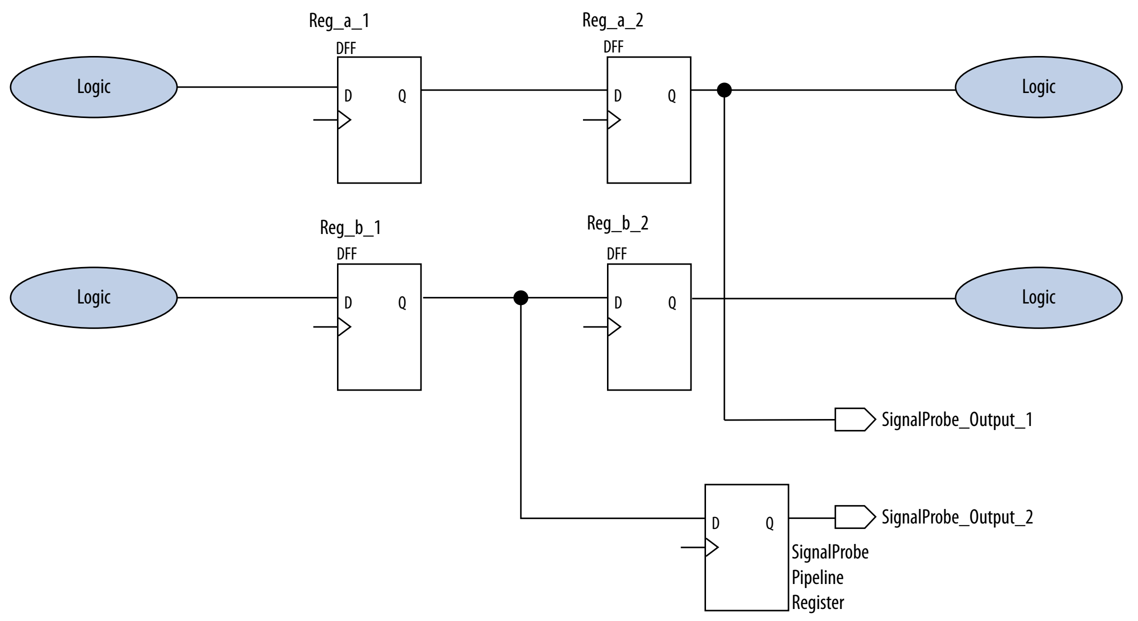
Add Registers Between Pipeline Paths and SignalProbe Pins

You can specify the number of registers placed between a SignalProbe source and a SignalProbe pin. The registers synchronize data to a clock and control the latency of the SignalProbe outputs. The SignalProbe feature automatically inserts the number of registers specified into the SignalProbe path.

The figure shows a single register between the SignalProbe source `Reg_b_1` and SignalProbe `SignalP-robe_Output_2` output pin added to synchronize the data between the two SignalProbe output pins.

Note: When you add a register to a SignalProbe pin, the SignalProbe compilation attempts to place the register to best meet timing requirements. You can place SignalProbe registers either near the SignalProbe source to meet f_{MAX} requirements, or near the I/O to meet t_{CO} requirements.

Figure 8-1: Synchronizing SignalProbe Outputs with a SignalProbe 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 Information

Add SignalProbe Pins Dialog Box online help
Information about how to pipeline an existing SignalProbe connection

Perform a SignalProbe Compilation

Perform a SignalProbe compilation to route your SignalProbe pins. A SignalProbe compilation saves and checks all netlist changes without recompiling the other parts of the design. A SignalProbe compilation takes a fraction of the time of a full compilation to finish. The design's current placement and routing are preserved.

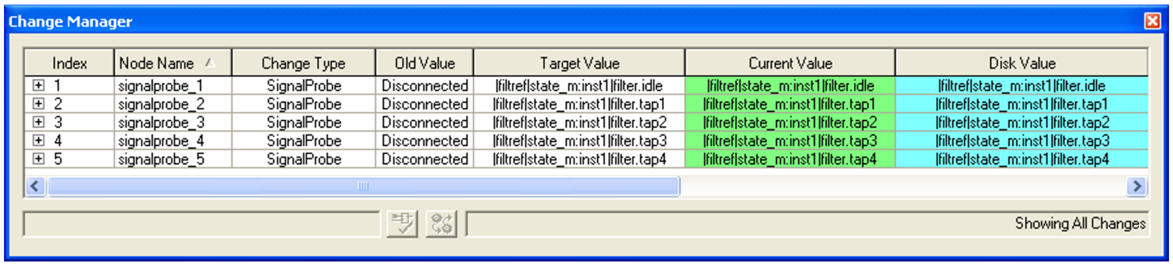
To perform a SignalProbe compilation, on the Processing menu, point to **Start** and click **Start SignalP-robe Compilation**.

Analyze the Results of a SignalProbe Compilation

After a SignalProbe compilation, the results are available in the compilation report file. Each SignalProbe pin is displayed in the **SignalProbe Fitting Result** page in the **Fitter** section of the Compilation Report. To view the status of each SignalProbe pin in the **SignalProbe Pins** dialog box, on the Tools menu, click **SignalProbe Pins**.

The status of each SignalProbe 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 8-2: Change Manager Window with SignalProbe 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 SignalProbe pin, on the Processing menu, point to **Start** and click **Start Timing Analysis**.

Related Information

Engineering Change Management with the Chip Planner documentation

What a SignalProbe Compilation Does

After a full compilation, you can start a SignalProbe compilation either manually or automatically. A SignalProbe compilation performs the following functions:

- Validates SignalProbe pins
- Validates your specified SignalProbe sources
- Adds registers into SignalProbe paths, if applicable
- Attempts to route from SignalProbe sources through registers to SignalProbe pins

To run the SignalProbe compilation immediately after a full compilation, on the Tools menu, click **SignalProbe Pins**. In the **SignalProbe Pins** dialog box, click **Start Check & Save All Netlist Changes**.

To run a SignalProbe compilation manually after a full compilation, on the Processing menu, point to **Start** and click **Start SignalProbe Compilation**.

Note: You must run the Fitter before a SignalProbe compilation. The Fitter generates a list of all internal nodes that can serve as SignalProbe sources.

Turn the **SignalProbe enable** option on or off in the **SignalProbe Pins** dialog box to enable or disable each SignalProbe pin.

Understanding the Results of a SignalProbe Compilation

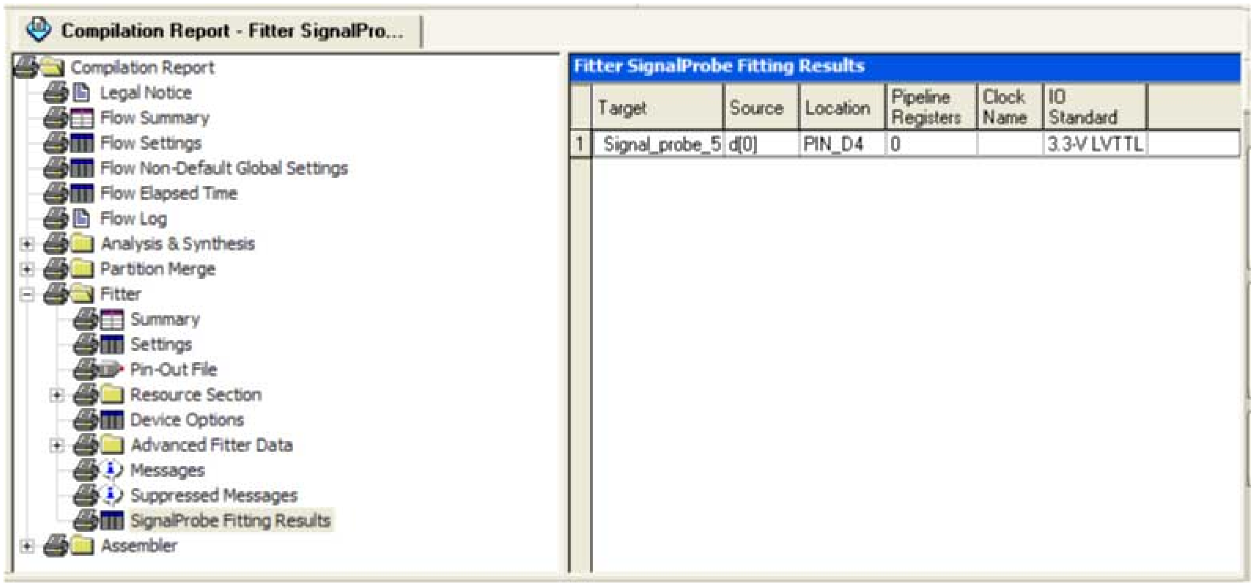
After a SignalProbe compilation, the results appear in two sections of the compilation report file. The fitting results and status of each SignalProbe pin appears in the **SignalProbe Fitting Result** screen in the Fitter section of the Compilation Report.

Table 8-1: Status Values

Status	Description
Routed	Connected and routed successfully
Not Routed	Not enabled

Status	Description
Failed to Route	Failed routing during last SignalProbe compilation
Need to Compile	Assignment changed since last SignalProbe compilation

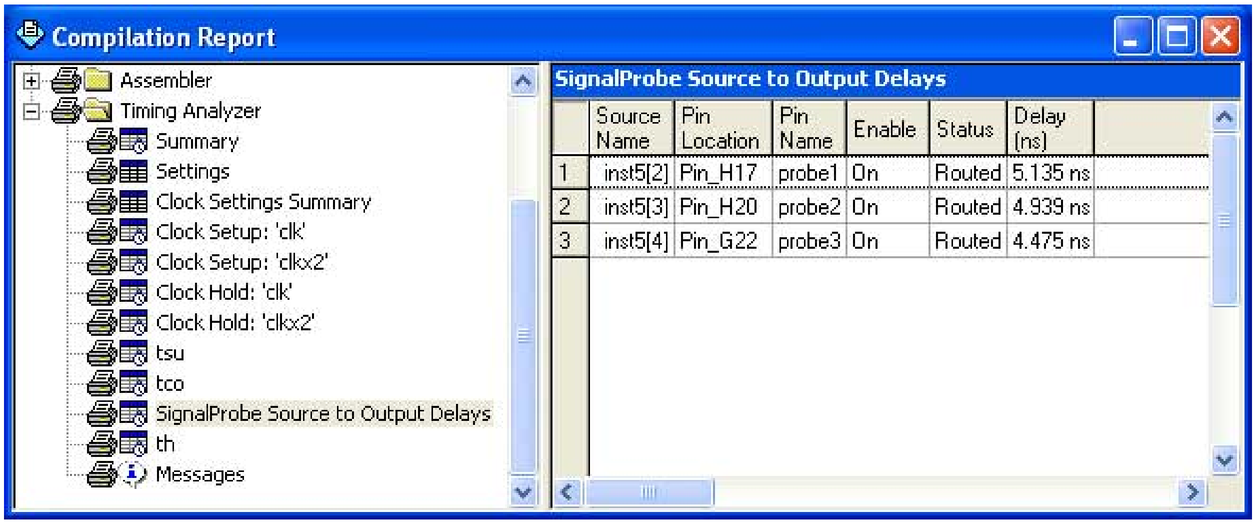
Figure 8-3: SignalProbe Fitting Results Page in the Compilation Report Window



	Target	Source	Location	Pipeline Registers	Clock Name	IO Standard
1	Signal_probe_5	d[0]	PIN_D4	0		3.3-V LVTTL

The **SignalProbe source to output delays** screen in the Timing Analysis section of the Compilation Report displays the timing results of each successfully routed SignalProbe pin.

Figure 8-4: SignalProbe Source to Output Delays Page in the Compilation Report Window



	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

Note: After a SignalProbe compilation, the processing screen of the Messages window also provides the results for each SignalProbe pin and displays slack information for each successfully routed SignalProbe pin.

Analyzing SignalProbe Routing Failures

A SignalProbe compilation can fail for any of the following reasons:

- **Route unavailable**—the SignalProbe compilation failed to find a route from the SignalProbe source to the SignalProbe pin because of routing congestion.
- **Invalid or nonexistent SignalProbe source**—you entered a SignalProbe 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 SignalProbe pin's I/O standard conflicts with other I/O standards in the same I/O bank.

If routing congestion prevents a successful SignalProbe compilation, you can allow the compiler to modify routing to the specified SignalProbe source. On the Tools menu, click **SignalProbe Pins** and turn on **Modify latest fitting results during SignalProbe compilation**. This setting allows the Fitter to modify existing routing channels used by your design.

Note: Turning on **Modify latest fitting results during SignalProbe compilation** can change the performance of your design.

Scripting Support

You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the 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` Quartus Prime Tcl API. Source or include the `::quartus::chip_planner` Tcl package in your scripts to make these commands available.

Related Information

- **Tcl Scripting documentation**
- **Quartus Prime Settings File Reference Manual**
  Information about all settings and constraints in the Quartus Prime software
- **Command-Line Scripting documentation**

Making a SignalProbe Pin

To make a SignalProbe 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>
```

Deleting a SignalProbe Pin

To delete a SignalProbe pin, type the following Tcl command:

```
delete_sp [-h | -help] [-long_help] -pin_name <pin name>
```

Enabling a SignalProbe Pin

To enable a SignalProbe pin, type the following Tcl command:

```
enable_sp [-h | -help] [-long_help] -pin_name <pin name>
```

Disabling a SignalProbe Pin

To disable a SignalProbe pin, type the following Tcl command:

```
disable_sp [-h | -help] [-long_help] -pin_name <pin name>
```

Performing a SignalProbe Compilation

To perform a SignalProbe compilation, type the following command:

```
quartus_sh --flow signalprobe <project name>
```

Script Example

The example shows a script that creates a SignalProbe pin called `sp1` and connects the `sp1` pin to source node `reg1` in a project that was already compiled.

Creating a SignalProbe 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
```

Reserving SignalProbe Pins

To reserve a SignalProbe pin, add the commands shown in the example to the Quartus Prime Settings File (**.qsf**) for your project.

Reserving a SignalProbe Pin

```
set_location_assignment <location> -to <SignalProbe pin name>
set_instance_assignment -name RESERVE_PIN \
"AS SIGNALPROBE OUTPUT" -to <SignalProbe pin name>
```

Valid locations are pin location names, such as `Pin_A3`.

Common Problems When Reserving a SignalProbe Pin

If you cannot reserve a SignalProbe pin in the 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 Quartus Prime Lite Edition software, in which the SignalProbe feature is not enabled by default. You must turn on TalkBack to enable the SignalProbe feature in the Quartus Prime Lite Edition software.
- 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 SignalProbe Output**.
- The pin is reserved from a previous compilation. During a compilation, the Quartus Prime software reserves each pin on the targeted device. If you end the 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 SignalProbe outputs.
- The pin does not support the SignalProbe feature. Select another pin.
- The current device family does not support the SignalProbe feature.

Adding SignalProbe Sources

To assign the node name to a SignalProbe pin, type the following Tcl command:

```
set_instance_assignment -name SIGNALPROBE_SOURCE <node name> \
-to <SignalProbe pin name>
```

The next command turns on SignalProbe routing. To turn off individual SignalProbe pins, specify `OFF` instead of `ON` with the following command:

```
set_instance_assignment -name SIGNALPROBE_ENABLE ON \
-to <SignalProbe pin name>
```

Related Information

- SignalProbe Pins Dialog Box online help
- Add SignalProbe Pins Dialog Box online help
  Information about how to pipeline an existing SignalProbe connection

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 <SignalProbe pin name>
```

Related Information

I/O Standards online help

Adding Registers for Pipelining

To add registers for pipelining, type the following Tcl command:

```
set_instance_assignment -name SIGNALPROBE_CLOCK <clock name> \
-to <SignalProbe pin name>

set_instance_assignment \
-name SIGNALPROBE_NUM_REGISTERS <number of registers> -to <SignalProbe pin name>
```

Running SignalProbe Immediately After a Full Compilation

To run SignalProbe immediately after a full compilation, type the following Tcl command:

```
set_global_assignment -name SIGNALPROBE_DURING_NORMAL_COMPILATION ON
```

Running SignalProbe Manually

To run SignalProbe 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>
```

Enabling or Disabling All SignalProbe Routing

Use the Tcl command in the example to turn on or turn off SignalProbe routing. When using this command, to turn SignalProbe routing on, specify `ON`. To turn SignalProbe routing off, specify `OFF`.

Turning SignalProbe 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> }
```

Allowing SignalProbe to Modify Fitting Results

To turn on **Modify latest fitting results**, type the following Tcl command:

```
set_global_assignment -name SIGNALPROBE_ALLOW_OVERUSE ON
```

Document Revision History

Table 8-2: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Quartus Prime*.
June 2014	14.0.0	Dita conversion.
May 2013	13.0.0	Changed sequence of flow to clarify that you need to perform a full compilation before reserving SignalProbe pins. Affected sections are “Debugging Using the SignalProbe Feature” on page 12–1 and “Reserving SignalProbe Pins” on page 12–2. Moved “Performing a Full Compilation” on page 12–2 before “Reserving SignalProbe Pins” on page 12–2.
June 2012	12.0.0	Removed survey link.

Date	Version	Changes
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 SignalProbe ECO flows • Removed support for SignalProbe pin preservation when recompiling with incremental compilation turned on. • Removed outdated FAQ section. • Added links to 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 SignalProbe connections when using Incremental Compilation • Added plausible scenarios where SignalProbe 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 Information

Quartus Handbook Archive

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.

Design Debugging Using the SignalTap II Logic Analyzer

9

2015.11.02

QPP5V3 Subscribe Send Feedback

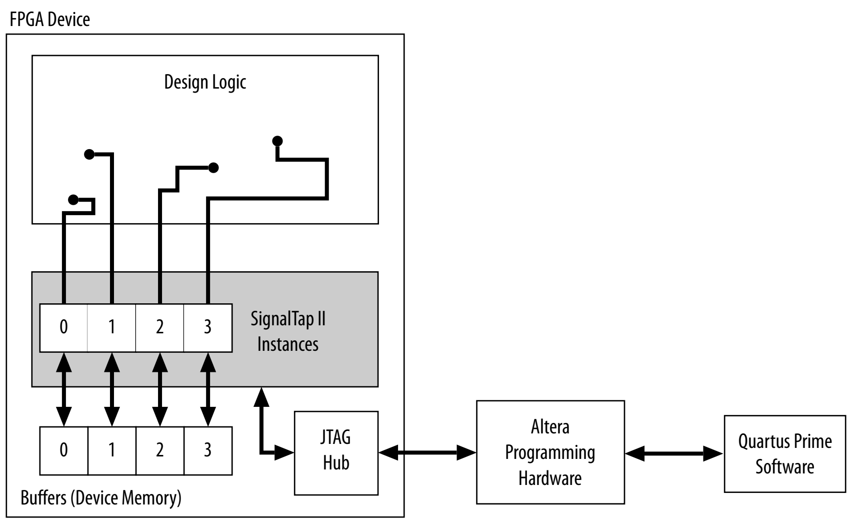
About the SignalTap II Logic Analyzer

The SignalTap® II 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 SignalTap II Logic Analyzer is scalable, easy to use, and available as a stand-alone package or with a software subscription. You can debug an FPGA design by probing the state of the design's internal signals without external equipment. Define custom trigger-condition logic for greater accuracy and improved ability to isolate problems. The SignalTap II Logic Analyzer does not require external probes or design file changes to capture the state of the internal nodes or I/O pins in the design. All captured signal data is conveniently stored in device memory until you are ready to read and analyze the data.

The SignalTap II 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 9-1: SignalTap II Logic Analyzer Block Diagram



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 SignalTap II Logic Analyzer is similar to traditional external logic analyzers, familiarity with external logic analyzer operations is helpful, but not necessary.

Note: The Quartus Prime Pro Edition software uses a new methodology for settings and assignments. For example, SignalTap II assignments include only the `instance` name, not the `entity:instance` name. Refer to Migrating to Quartus Prime Pro Edition for more information on migrating existing **.stp** files to Quartus Prime Pro Edition.

Related Information

Migrating to Quartus Prime Pro Edition
Migrating to the Quartus Prime Pro Edition software.

Hardware and Software Requirements

You need the following hardware and software to perform logic analysis with the SignalTap II Logic Analyzer:

- SignalTap software
- Download/upload cable
- Altera® development kit or your design board with JTAG connection to device under test

Use the SignalTap software that is included with the following software:

- Quartus Prime design software
- Quartus Prime Lite Edition (with the TalkBack feature enabled)

Alternatively, use the SignalTap II Logic Analyzer standalone software and standalone Programmer software.

Note: The Quartus Prime Lite Edition software does not support incremental compilation integration with the SignalTap II Logic Analyzer.

The memory blocks of the device store captured data. The memory blocks transfer the data to the Quartus Prime software waveform display over a JTAG communication cable, such as EthernetBlaster or USB-Blaster®.

Table 9-1: SignalTap II Logic Analyzer Features and Benefits

Feature	Benefit
Quick access toolbar	Single-click operation of commonly used menu items. Hover over the icons to see tool tips.
Multiple logic analyzers in a single device	Captures data from multiple clock domains in a design at the same time.
Multiple logic analyzers in multiple devices in a single JTAG chain	Simultaneously captures data from multiple devices in a JTAG chain.
Nios II plug-in support	Easily specifies nodes, triggers, and signal mnemonics for IP, such as the Nios® II processor.
Up to 10 basic or advanced trigger conditions for each analyzer instance	Enables sending more complex data capture commands to the logic analyzer, providing greater accuracy and problem isolation.

Feature	Benefit
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 route with rapid recompile	Manually allocate trigger input, data input, storage filter input node count, and perform a full compilation to include the SignalTap II 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 SignalTap II Logic Analyzer captured data 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 in each device	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 estimate of logic and memory device resources used by SignalTap II Logic Analyzer configurations.
No additional cost	The SignalTap II Logic Analyzer is included with a Quartus Prime subscription and with the Quartus Prime Lite Edition (with TalkBack enabled).
Compatibility with other on-chip debugging utilities	You can use the SignalTap II 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 SignalTap II Logic Analyzer.
Floating-Point Display Format	To enable, click **Edit** > **Bus Display Format** > **Floating-point** Supports the following: • Single-precision floating-point format **IEEE754 Single (32-bit)** • Double-precision floating-point format **IEEE754 Double (64-bit)**

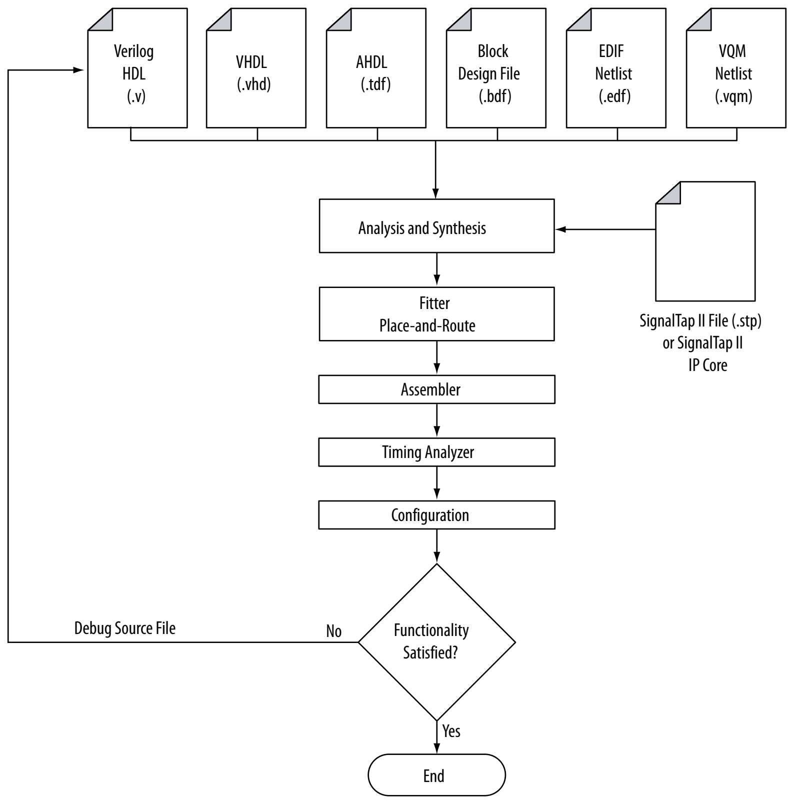
Related Information

- **System Debugging Tools Overview documentation** on page 5-1
  Overview and comparison of all tools available in the In-System Verification Tool set

Design Flow Using the SignalTap II Logic Analyzer

Add a SignalTap II file (**.stp**) and enable it in your project, or instantiate a SignalTap II IP core created with the parameter editor. **Figure 9-2** shows the flow of operations from initially adding the SignalTap II Logic Analyzer to your design to final device configuration, testing, and debugging.

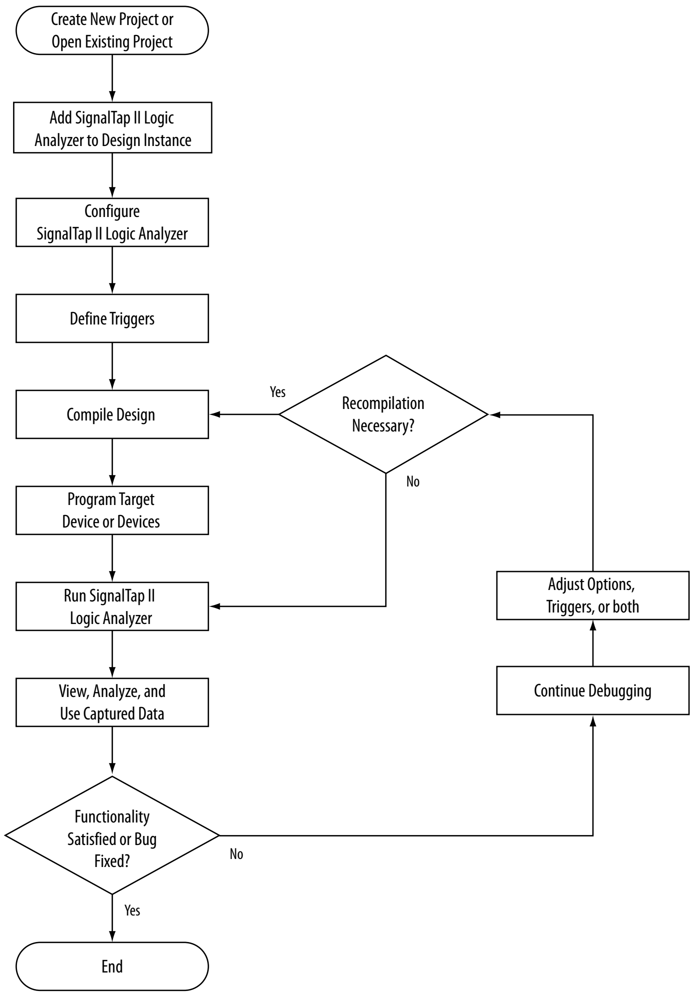
Figure 9-2: SignalTap II FPGA Design and Debugging Flow



SignalTap II Logic Analyzer Task Flow Overview

To use the SignalTap II Logic Analyzer to debug your design, you perform a number of tasks to add, configure, and run the logic analyzer.

Figure 9-3: SignalTap II Logic Analyzer Task Flow


Create New Project or Open Existing Project
Add SignalTap II Logic Analyzer to Design Instance
Configure SignalTap II Logic Analyzer
Define Triggers
Compile Design
Yes
Recompilation Necessary?
No
Program Target Device or Devices
Run SignalTap II Logic Analyzer
Adjust Options, Triggers, or both
View, Analyze, and Use Captured Data
Continue Debugging
Functionality Satisfied or Bug Fixed?
No
Yes
End

Add the SignalTap II 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.

Configure the SignalTap II Logic Analyzer

After you add the SignalTap II Logic Analyzer to your design, configure the logic analyzer to monitor the signals you want. You can manually add signals or use a plug-in, such as the Nios II processor plug-in, to quickly add entire sets of associated signals for a particular intellectual property (IP). You can also specify settings for the data capture buffer, such as its size, the method in which data is captured and stored, and the device memory type to use for the buffer in devices that support memory type selection.

Related Information

Creating a Power-Up Trigger on page 9-42

Define Trigger Conditions

The SignalTap II Logic Analyzer captures data continuously while the logic analyzer is running. To capture and store specific signal data, set up triggers that tell the logic analyzer under what conditions to stop capturing data. The SignalTap II 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.

Compile the Design

With the **.stp** configured and trigger conditions defined, compile your project as usual to include the logic analyzer in your design.

Program the Target Device or Devices

When you debug a design with the SignalTap II Logic Analyzer, you can program a target device directly from the **.stp** without using the Quartus Prime Programmer. You can also program multiple devices with different designs and simultaneously debug them.

Note: The SignalTap II Logic Analyzer supports all current Altera FPGA device families.

Related Information

Managing Multiple SignalTap II Files and Configurations on page 9-21

Run the SignalTap II 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 Information

Analyzing Data in the SignalTap II Logic Analyzer online help

View, Analyze, and Use Captured Data

After you have captured data and read it into the **.stp**, that data is available for analysis and debugging. Set up mnemonic tables, either manually or with a plug-in, to simplify reading and interpreting the captured signal data. To speed up debugging, use the **Locate** feature in the **SignalTap II node** list to find the locations of problem nodes in other tools in the Quartus Prime software. Save the captured data for later analysis, or convert the data to other formats for sharing and further study.

Embed Multiple Analyzers in One FPGA

The SignalTap II Logic Analyzer Editor includes support for adding multiple logic analyzers by creating instances in the **.stp**. You can create a unique logic analyzer for each clock domain in the design.

Monitor FPGA Resources Used by the SignalTap II Logic Analyzer

The SignalTap II 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 of each logic analyzer instance and total resources used in the columns of the **Instance Manager** pane of the SignalTap II Logic Analyzer Editor. Use this feature when you know that your design is running low on resources.

The logic element value reported in the resource usage estimator may vary by as much as 10% from the actual resource usage.

Use the Parameter Editor to Create Your Logic Analyzer

You can create a SignalTap II Logic Analyzer instance by using the parameter editor. The parameter editor generates an HDL file that you instantiate in your design.

Note: The state-based trigger flow, the state machine debugging feature, and the storage qualification feature are not supported when using the parameter editor to create the logic analyzer.

Configuring the SignalTap II Logic Analyzer

There are several ways to configure instances of the SignalTap II Logic Analyzer. Some settings are similar to those found on traditional external logic analyzers. Other settings are unique to the SignalTap II Logic Analyzer because it is configurable.

Note: You can only adjust some settings when you are viewing run-time trigger conditions instead of power-up trigger conditions.

Assigning an Acquisition Clock

Assign a clock signal to control how the SignalTap II Logic Analyzer acquires data. 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, Altera recommends that you use a global, non-gated clock synchronous to the signals under test for data acquisition. Using a gated clock as your acquisition clock can result in unexpected data that does not accurately reflect the behavior of your design. The Quartus Prime static timing analysis tools show the maximum acquisition clock frequency at which you can run your design. Refer to the Timing Analysis section of the Compilation Report to find the maximum frequency of the logic analyzer clock.

Note: Exercise caution when using a recovered clock from a transceiver as an acquisition clock for the SignalTap II 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 SignalTap II Logic Analyzer Editor, the Quartus Prime software automatically creates a clock pin called `auto_stp_external_clk`. You must make a pin assignment to this pin and ensure that a clock signal in your design drives the acquisition clock.

Related Information

Managing Device I/O Pins documentation
Information about assigning signals to pins

Adding Signals to the SignalTap II File

Add signals to the **.stp** node list to select which signals in your design you want to monitor. You can also select signals to define triggers. You can assign the following two signal types:

- **Pre-synthesis**—These signals exists after design elaboration, but before any synthesis optimizations are done. This set of signals should reflect your Register Transfer Level (RTL) signals.
- **Post-fitting**—This signal exists after physical synthesis optimizations and place-and-route.

The Quartus Prime software does not limit the number of signals available for monitoring in the SignalTap II 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.

Note: The Quartus Prime Pro Edition software uses only the instance name, and not the entity name, in the form of:

`a|b|c`

not `a_entity:a|b_entity:b|c_entity:c`

After successful Analysis and Elaboration, invalid signals are displayed in red. Unless you are certain that these signals are valid, remove them from the **.stp** for correct operation. The SignalTap II Status Indicator also indicates if an invalid node name exists in the **.stp**.

You can tap signals if a routing resource (row or column interconnects) exists to route the connection to the SignalTap II instance. For example, signals that exist in the I/O element (IOE) cannot be directly tapped 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.

When adding pre-synthesis signals, make all connections to the SignalTap II Logic Analyzer before synthesis. Logic and routing resources are allocated during recompilation to make the connection as if a change in your design files had been made. Pre-synthesis signal names for signals driving to and from IOEs coincide with the signal names assigned to the pin.

In the case of post-fit signals, connections that you make to the SignalTap II 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 SignalTap II 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 signal name assigned to the pin.

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. The Technology Map viewer and the Resource Property Editor can also be used to help you find post-fit node names.

Related Information

Analyzing Designs with Quartus Prime Netlist Viewers documentation
Information about cross-probing to source design files and other Quartus Prime windows

Preserving Signals

The Quartus Prime software optimizes the RTL signals during synthesis and place-and-route. RTL signal names frequently 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. When the Quartus Prime software encounters the following synthesis attributes, it does not perform any optimization on the specified signals, allowing them to persist into the post-fit netlist.

- `keep`—Ensures that combinational signals are not removed
- `preserve`—Ensures that registers are not removed

Using these attributes can increase device resource utilization or decrease timing performance.

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 make them available for debugging with the SignalTap II Logic Analyzer. Preserving nodes is often necessary when a plug-in is used to add a group of signals for a particular IP.

Related Information

Quartus Prime Integrated Synthesis documentation
Information about using signal preservation attributes

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 compiling your design. When you find the desired node, copy the node to either the active **.stp** for your design or a new **.stp**.

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.

You can turn off the ability of a signal to trigger the analyzer by disabling the **Trigger Enable** option for that signal in the node list in the **.stp**. This option is useful when you want to see only the captured data for a signal and you are not using that signal as part of a trigger.

You can turn off the ability to view data for a signal by disabling the **Data Enable** column. This option is useful when you want to trigger on a signal, but have no interest in viewing data for that signal.

Related Information

Define Triggers on page 9-22

Disabling and Enabling a SignalTap II Instance

Disable and enable SignalTap II instances in the **Instance Manager** pane. Physically adding or removing instances requires recompilation after disabling and enabling a SignalTap II instance.

Untappable Signals

Not all of the post-fitting signals in your design are available in the **SignalTap II : post-fitting filter** in the **Node Finder** dialog box. The following signal types cannot be tapped:

- **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.

Adding Signals with a Plug-In

Instead of adding individual or grouped signals through the **Node Finder**, you can add groups of relevant signals of a particular type of IP with a plug-in. The SignalTap II Logic Analyzer comes with one plug-in already installed for the Nios II processor. Besides easy signal addition, plug-ins also 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 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)—Displays disassembled code from the corresponding address.

To add signals to the **.stp** using a plug-in, perform the following steps after running Analysis and Elaboration on your design:

1. Right-click in the node list. On the Add Nodes with Plug-In submenu, choose the plug-in you want to use, such as the included plug-in named **Nios II**.

   Note: 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.
2. The **Select Hierarchy Level** dialog box appears showing the IP hierarchy of your design. Select the IP that contains the signals you want to monitor with the plug-in and click **OK**.
3. If all the signals in the plug-in are available, a dialog box might appear, depending on the plug-in selected, where you can specify options for the plug-in. 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 as desired and click **OK**.

   Note: To make sure all the required signals are available, in the Quartus Prime **Analysis & Synthesis** settings, click 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.

Related Information

- **Define Triggers** on page 9-22
- **View, Analyze, and Use Captured Data** on page 9-6

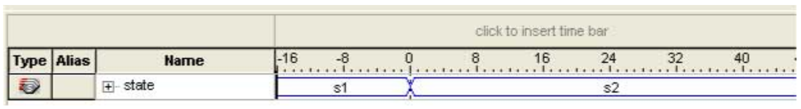
Adding Finite State Machine State Encoding Registers

Finding the signals to debug finite state machines (FSM) can be challenging. Finding nodes from the post-fit netlist may be impossible, as FSM encoding signals may be changed or optimized away during synthesis and place-and-route. If you can find all of the relevant nodes in the post-fit netlist or you used the nodes from the pre-synthesis netlist, an additional step is required to find and map FSM signal values to the state names that you specified in your HDL.

The SignalTap II Logic Analyzer can detect FSMs in your compiled design. The SignalTap II Logic Analyzer configuration automatically tracks the FSM state signals as well as state encoding through the compilation process. Shortcut menu commands allow you to add all of the FSM state signals to your logic analyzer with a single command. For each FSM added to your SignalTap II 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 9-4: Decoded FSM Mnemonics

The waveform viewer with decoded signal values from a state machine added with the FSM debugging feature.



Related Information

Recommended HDL Coding Styles documentation
Coding guidelines for specifying FSM in Verilog HDL and VHDL

Modifying and Restoring Mnemonic Tables for State Machines

When you add FSM state signals via the FSM debugging feature, the SignalTap II 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. You can edit any mnemonic table using the **Mnemonic Table Setup** dialog box.

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 Information

Creating Mnemonics for Bit Patterns on page 9-54

Additional Considerations

The SignalTap II configuration GUI recognizes state machines from your design only if you use Quartus Prime Integrated Synthesis (QIS). 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 SignalTap II 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 SignalTap II FSM debug feature may not list mnemonic tables for state machines in the design:

- If you have physical synthesis turned on, 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 have been packed into RAM and DSP blocks during QIS or Fitter optimizations.

You can still use the FSM debugging feature to add pre-synthesis state signals.

Specifying the Sample Depth

The sample depth specifies the number of samples that are captured and stored for each signal in the captured data buffer. To specify the sample depth, select the desired number of samples to store in the **Sample Depth** list. 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.

Capturing Data to a Specific RAM Type

When you use the SignalTap II Logic Analyzer with some devices, you have the option to select the RAM type where acquisition data is stored. Once SignalTap II Logic Analyzer is allocated 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 SignalTap II Logic Analyzer data acquisition. For example, if your design has an application that requires a large block of memory resources, such a large instruction or data cache, you would choose to use MLAB, M512, or M4k blocks for data acquisition and leave the M9k blocks for the rest of your design.

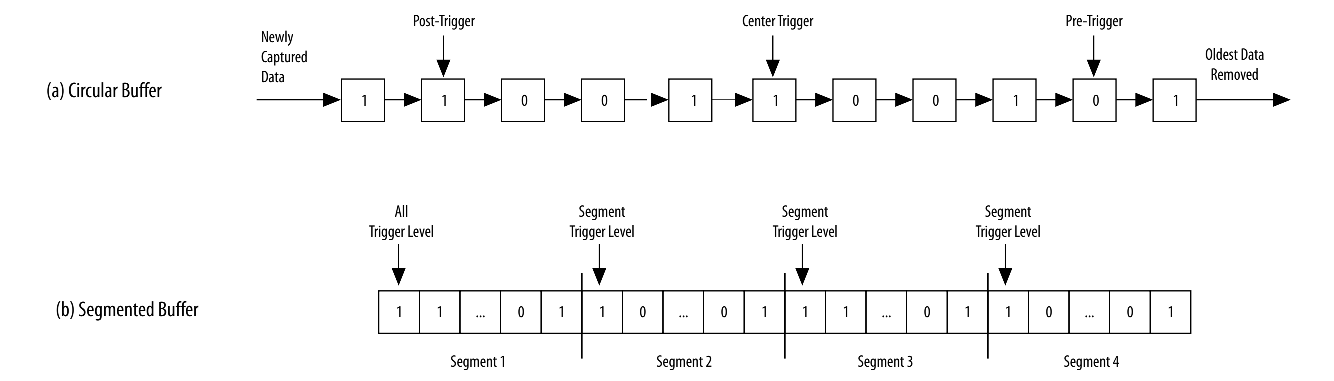
To select the RAM type to use for the SignalTap II Logic Analyzer buffer, select it from the RAM type list. Use this feature when the acquired data (as reported by the SignalTap II resource estimator) is not larger than the available memory of the memory type that you have selected in the FPGA.

Choosing the Buffer Acquisition Mode

The Buffer Acquisition Type Selection feature in the SignalTap II Logic Analyzer lets you choose how the captured data buffer is organized and can potentially reduce the amount of memory that is required for SignalTap II data acquisition. There are two types of acquisition buffer within the SignalTap II Logic Analyzer—a non-segmented (or circular) buffer and a segmented buffer. With a non-segmented buffer, the SignalTap II 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 a number of separate buffers. Each buffer acts as a separate FIFO with its own set of trigger conditions. Only a single buffer is active during an acquisition. The SignalTap II 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 9-5 illustrates the differences between the two buffer types.

Figure 9-5: Buffer Type Comparison in the SignalTap II Logic Analyzer



Notes to figure :

1. Both non-segmented and segmented buffers can use a predefined trigger (Pre-Trigger, Center Trigger, Post-Trigger) position or define a custom trigger position using the **State-Based Triggering** tab. Refer to **Specifying the Trigger Position** for more details.
2. Each segment is treated like a FIFO, and behaves as the non-segmented buffer shown in (a).

Related Information

Using the Storage Qualifier Feature on page 9-14

Non-Segmented Buffer

The non-segmented buffer (also known as a circular buffer) shown in **Figure 9-5** (a) is the default buffer type used by the SignalTap II Logic Analyzer. While the logic analyzer is running, data is stored in the buffer until it fills up, at which point new data replaces the oldest data. This continues until a specified trigger event, consisting of a set of trigger conditions, occurs. When the trigger event happens, the logic analyzer continues to capture data after the trigger event until the buffer is full, based on the trigger position setting in the **Signal Configuration** pane or the **.stp**. To capture the majority of the data before the trigger occurs, select **Post trigger position** from the list. To capture the majority of the 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.

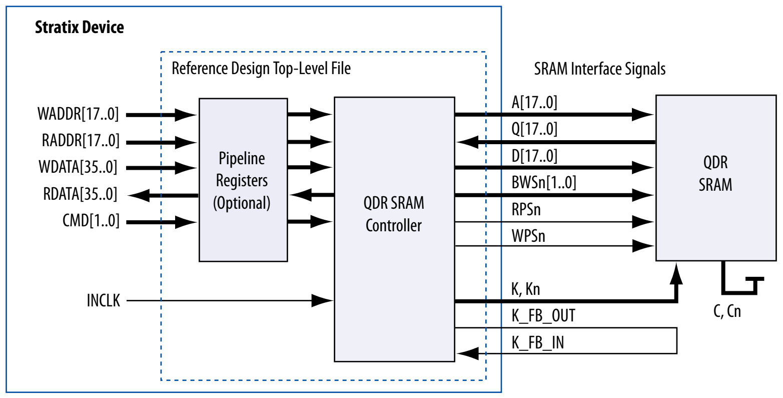
Related Information

Specifying the Trigger Position on page 9-41

Segmented Buffer

A segmented buffer allows you to debug systems that contain relatively infrequent recurring events. The acquisition memory is split into evenly sized segments, with a set of trigger conditions defined for each segment. Each segment acts as a non-segmented buffer. If you want to have separate trigger conditions for each of the segmented buffers, you must use the state-based trigger flow. **Figure 9-6** shows an example of a segmented buffer system. If you want to have separate trigger conditions for each of the segmented buffers, you must use the state-based trigger flow.

Figure 9-6: Example System that Generates Recurring Events



The SignalTap II Logic Analyzer verifies the functionality of the design shown in **Figure 9-6** to ensure that the correct data is written to the SRAM controller. Buffer acquisition in the SignalTap II Logic Analyzer allows you to monitor the `RDATA` port when `H'0F0F0F0F` is sent into the `RADDR` port. You can monitor multiple read transactions from the SRAM device without running the SignalTap II Logic Analyzer again. The buffer acquisition feature allows you to segment the memory so you can capture the same event multiple times without wasting allocated memory. The number of cycles that are captured depends on the number of segments specified under the **Data** settings.

To enable and configure buffer acquisition, select **Segmented** in the SignalTap II Logic Analyzer Editor and select the number of segments to use. In the example in **Figure 9-6**, selecting sixty-four 64-sample segments allows you to capture 64 read cycles when the `RADDR` signal is `H'0F0F0F0F`.

Using the Storage Qualifier Feature

Both non-segmented and segmented buffers described in the previous section offer a snapshot in time of the data stream being analyzed. The default behavior for writing into acquisition memory with the SignalTap II Logic Analyzer is to sample data on every clock cycle. With a non-segmented buffer, there is one data window that represents a comprehensive snapshot of the datastream. Similarly, segmented buffers use several smaller sampling windows spread out over more time, with each sampling window representing a contiguous data set.

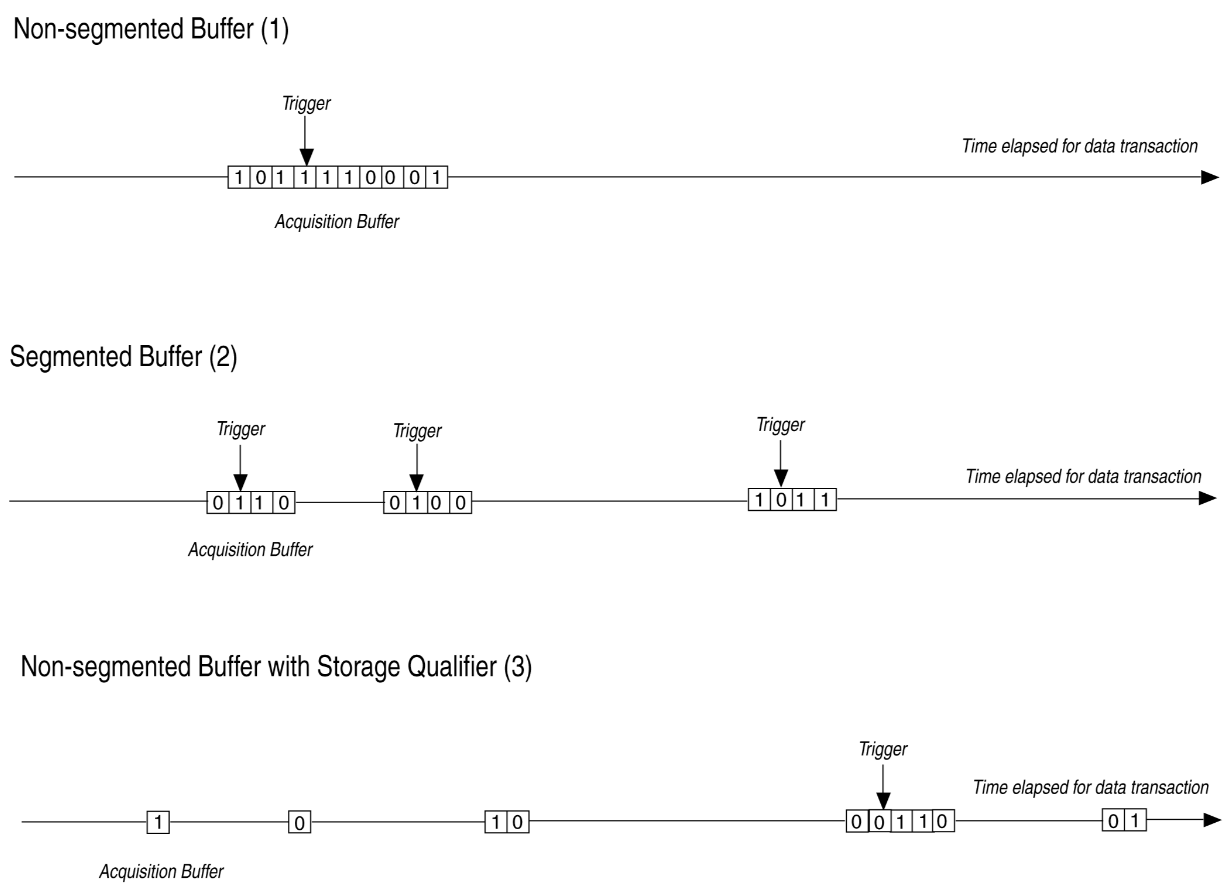
With carefully chosen trigger conditions and a generous sample depth for the acquisition buffer, analysis using segmented and non-segmented buffers captures a majority of functional errors in a chosen signal set. However, each data window can have a considerable amount of redundancy associated with it; for example, a capture of a data stream containing long periods of idle signals between data bursts. With default behavior using the SignalTap II Logic Analyzer, you cannot discard the redundant sample bits.

The Storage Qualification feature allows you to filter out individual samples not relevant to debugging the design. With this feature, a condition acts as a write enable to the buffer during each clock cycle of data acquisition. Through fine tuning the data that is actually stored in acquisition memory, the Storage Qualification feature allows for a more efficient use of acquisition memory in the specified number of samples over a longer period of analysis.

Use of the Storage Qualification feature is similar to an acquisition using a segmented buffer, in that you can create a discontinuity in the capture buffer. Because you can create a discontinuity between any two samples in the buffer, the Storage Qualification feature is equivalent to being able to create a customized segmented buffer in which the number and size of segment boundaries are adjustable.

Note: You can only use the Storage Qualification feature with a non-segmented buffer. The IP Catalog flow only supports the Input Port mode for the Storage Qualification feature.

Figure 9-7: 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 Qualification allows you to define a custom sampling window for each segment you create with a qualifying condition. Storage qualification potentially allows for a larger time scale of coverage.

There are six storage qualifier types available under the Storage Qualification feature:

- **Continuous**
- **Input port**
- **Transitional**
- **Conditional**
- **Start/Stop**
- **State-based**

Continuous (the default mode selected) turns the Storage Qualification feature off.

Each selected storage qualifier type is active when an acquisition starts. Upon the start of an acquisition, the SignalTap II Logic Analyzer examines each clock cycle and writes the data into the acquisition buffer based upon storage qualifier type and condition. The acquisition stops when a defined set of trigger conditions occur.

Note: Trigger conditions are evaluated independently of storage qualifier conditions. The SignalTap II Logic Analyzer evaluates the data stream for trigger conditions on every clock cycle after the acquisition begins.

The storage qualifier operates independently of the trigger conditions.

Related Information

Define Trigger Conditions on page 9-6

Input Port Mode

When using the Input port mode, the SignalTap II Logic Analyzer takes any signal from your design as an input. When the design is running, if the signal is high on the clock edge, the SignalTap II Logic Analyzer stores the data in the buffer. If the signal is low on the clock edge, the data sample is ignored. A pin is created and connected to this input port by default if no internal node is specified.

If you are using an **.stp** to create a SignalTap II Logic Analyzer instance, 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 9-8: Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (Segmented Buffer)

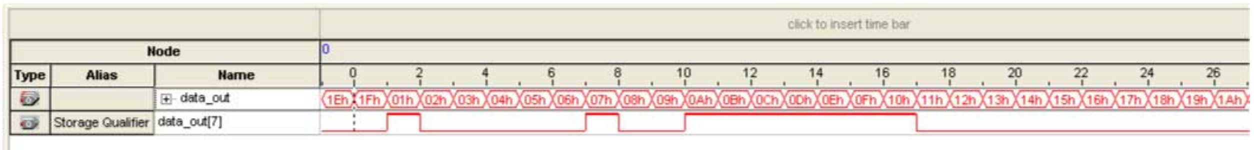
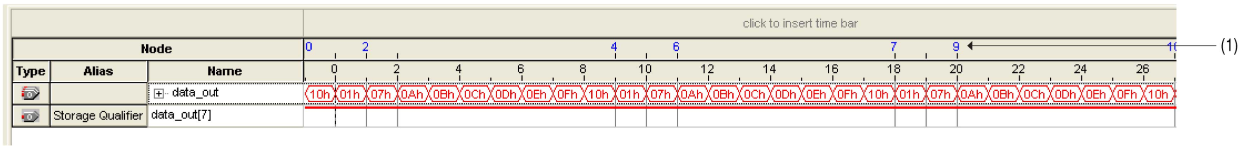


Figure 9-9: Data Acquisition of a Recurring Data Pattern Using an Input Signal as a Storage Qualifier

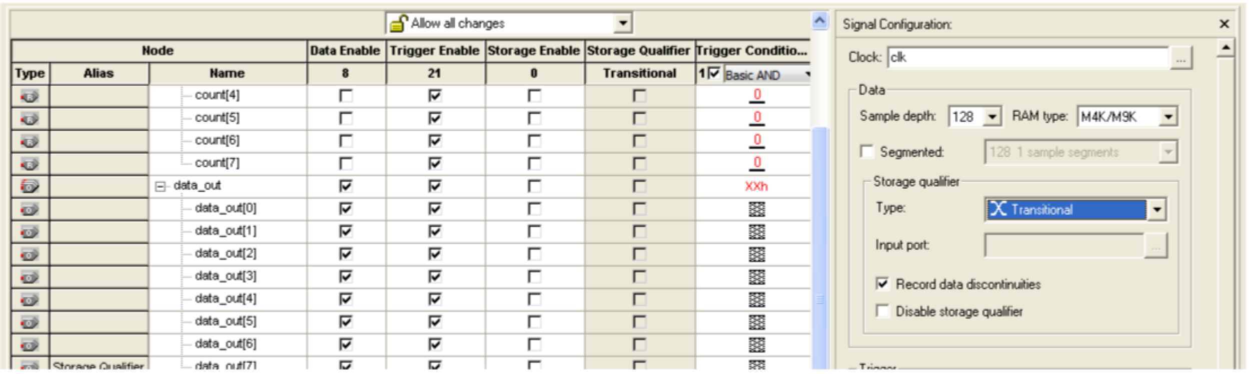


(1) Markers display samples when the logic analyzer paused a write into acquisition memory. These markers are enabled with the option "Record data discontinuities."

Transitional Mode

In Transitional mode, you choose a set of signals for inspection using the node list check boxes in the **Storage Qualifier** column. During acquisition, if any of the signals marked for inspection have changed since the previous clock cycle, new data is written to the acquisition buffer. If none of the signals marked have changed since the previous clock cycle, no data is stored.

Figure 9-10: Transitional Storage Qualifier Setup



Type	Alias	Name	Data Enable 8	Trigger Enable 21	Storage Enable 0	Storage Qualifier Transitional	Trigger Conditio... 1 ✓ Basic AND
		count[4]	☐	✓	☐	☐	0
		count[5]	☐	✓	☐	☐	0
		count[6]	☐	✓	☐	☐	0
		count[7]	☐	✓	☐	☐	0
		data_out	✓	✓	☐	☐	XXh
		data_out[0]	✓	✓	☐	☐	▨
		data_out[1]	✓	✓	☐	☐	▨
		data_out[2]	✓	✓	☐	☐	▨
		data_out[3]	✓	✓	☐	☐	▨
		data_out[4]	✓	✓	☐	☐	▨
		data_out[5]	✓	✓	☐	☐	▨
		data_out[6]	✓	✓	☐	☐	▨
	Storage Qualifier	data_out[7]	✓	✓	☐	☐	▨

Signal Configuration:

Clock: clk

Data

Sample depth: 128 RAM type: M4K/M9K

☐ Segmented: 128 1 sample segments

Storage qualifier

Type: Transitional

Input port:

☑ Record data discontinuities

☐ Disable storage qualifier

Figure 9-11: Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (Transitional Mode)

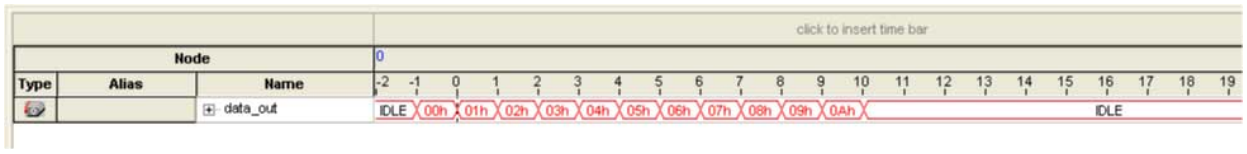
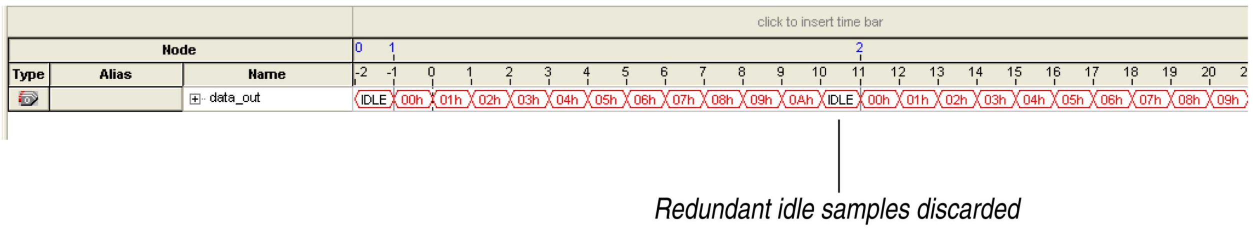


Figure 9-12: Data Acquisition of Recurring Data Pattern Using a Transitional Mode as a Storage Qualifier



Conditional Mode

In Conditional mode, the SignalTap II Logic Analyzer evaluates a combinational function of storage qualifier enabled signals within the node list to determine whether a sample is stored. The SignalTap II 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**, or **Advanced** storage qualifier conditions. A **Basic AND** or **Basic OR** storage qualifier condition matches each signal to one of the following:

- **Don’t Care**
- **Low**
- **High**
- **Falling Edge**
- **Rising Edge**
- **Either Edge**

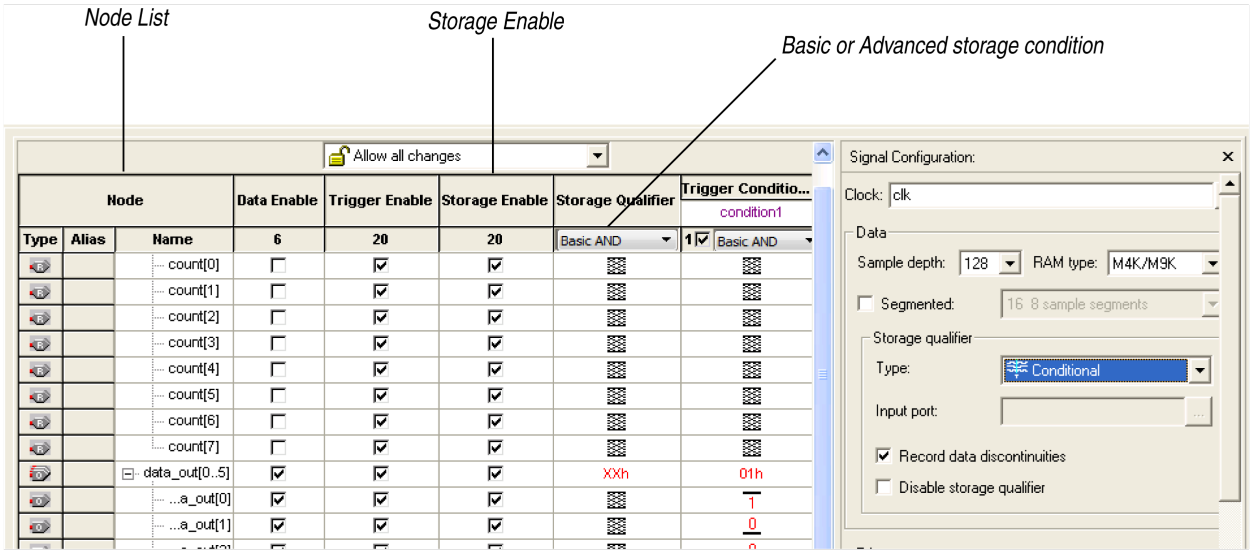
If you specify a **Basic AND** storage qualifier condition for more than one signal, the SignalTap II Logic Analyzer evaluates the logical AND of the conditions.

Any other combinational or relational operators that you may want to specify with the enabled signal set for storage qualification can be done with an advanced storage condition. **Figure 9-13** details the conditional storage qualifier setup in the **.stp**.

You can specify storage qualification conditions similar to the manner in which trigger conditions are specified.

Figure 9-14 and **Figure 9-15** show a data capture with continuous sampling, and the same data pattern using the conditional mode for analysis, respectively.

Figure 9-13: Conditional Storage Qualifier Setup



Type	Alias	Name	Data Enable	Trigger Enable	Storage Enable	Storage Qualifier	Trigger Conditio... condition1
			6	20	20	Basic AND	1 ☑ Basic AND
		count[0]	☐	☑	☑	▨	▨
		count[1]	☐	☑	☑	▨	▨
		count[2]	☐	☑	☑	▨	▨
		count[3]	☐	☑	☑	▨	▨
		count[4]	☐	☑	☑	▨	▨
		count[5]	☐	☑	☑	▨	▨
		count[6]	☐	☑	☑	▨	▨
		count[7]	☐	☑	☑	▨	▨
		⊟ data_out[0..5]	☑	☑	☑	XXh	01h
		...a_out[0]	☑	☑	☑	▨	1
		...a_out[1]	☑	☑	☑	▨	0

Signal Configuration:

Clock: clk

Data

Sample depth: 128 RAM type: M4K/M9K

☐ Segmented: 16 8 sample segments

Storage qualifier

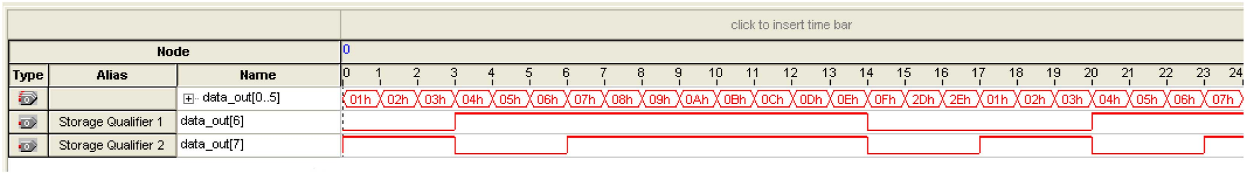
Type: Conditional

Input port:

☑ Record data discontinuities

☐ Disable storage qualifier

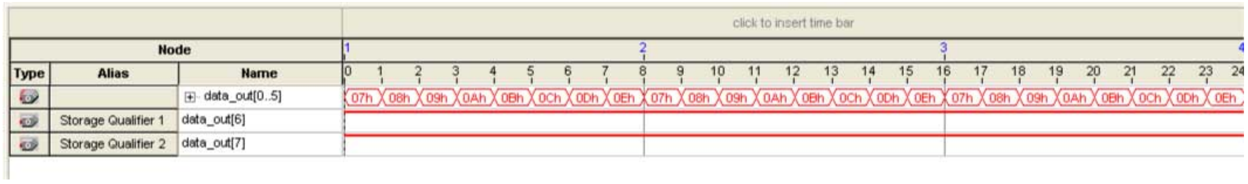
Figure 9-14: Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (Conditional)


click to insert time bar
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
01h 02h 03h 04h 05h 06h 07h 08h 09h 0Ah 0Bh 0Ch 0Dh 0Eh 0Fh 2Dh 2Eh 01h 02h 03h 04h 05h 06h 07h

Type	Alias	Name
		⊞ data_out[0..5]
	Storage Qualifier 1	data_out[6]
	Storage Qualifier 2	data_out[7]

(1) Storage Qualifier condition is set up to evaluate data_out[6] AND data_out[7].

Figure 9-15: Data Acquisition of a Recurring Data Pattern in Conditional Capture Mode


click to insert time bar
1 2 3 4
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
07h 08h 09h 0Ah 0Bh 0Ch 0Dh 0Eh 07h 08h 09h 0Ah 0Bh 0Ch 0Dh 0Eh 07h 08h 09h 0Ah 0Bh 0Ch 0Dh 0Eh

Type	Alias	Name
		⊞ data_out[0..5]
	Storage Qualifier 1	data_out[6]
	Storage Qualifier 2	data_out[7]

Related Information

- Creating Basic Trigger Conditions on page 9-23
- Creating Advanced Trigger Conditions on page 9-25

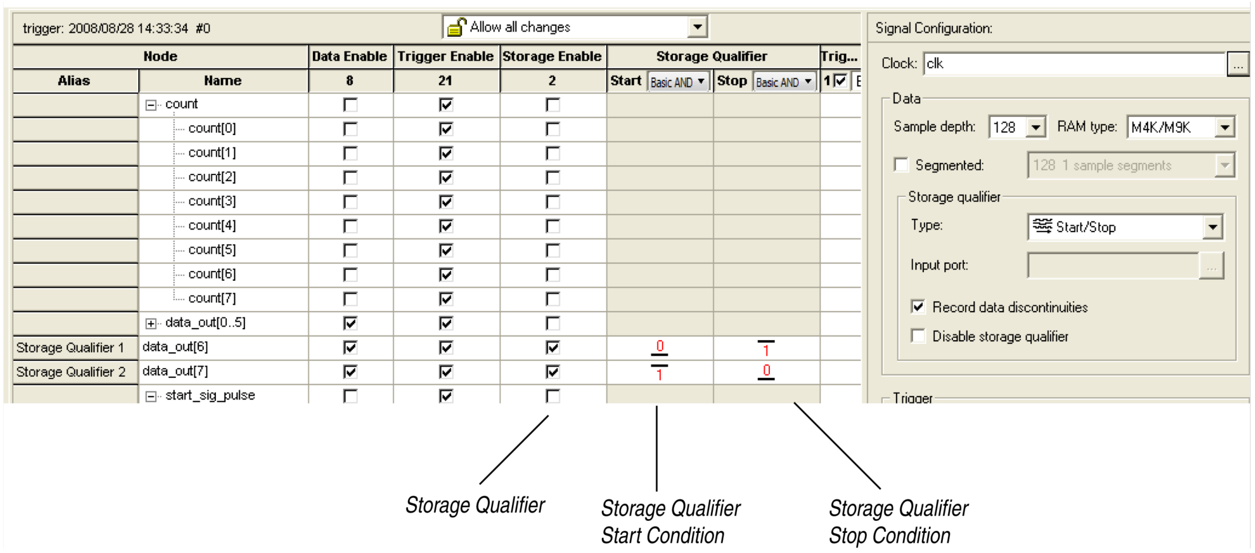
Start/Stop Mode

The Start/Stop mode is similar to the Conditional mode for storage qualification. However, in this mode there are two sets of conditions, one for start and one for stop. If the start condition evaluates to TRUE, data is stored in the buffer every clock cycle until the stop condition evaluates to TRUE, which then pauses the data capture. Additional start signals received after the data capture has started are ignored. If both start and stop evaluate to TRUE at the same time, a single cycle is captured.

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 9-16 shows the Start/Stop mode storage qualifier setup. Figure 9-17 and Figure 9-18 show a data capture pattern in continuous capture mode and a data pattern in using the Start/Stop mode for storage qualification.

Figure 9-16: Start/Stop Mode Storage Qualifier Setup


Storage Qualifier
Storage Qualifier Start Condition
Storage Qualifier Stop Condition

trigger: 2008/08/28 14:33:34 #0 — Allow all changes

Node		Data Enable	Trigger Enable	Storage Enable	Storage Qualifier		Trig...
Alias	Name	8	21	2	Start Basic AND	Stop Basic AND	1 ☑
	count	☐	☑	☐			
	count[0]	☐	☑	☐			
	count[1]	☐	☑	☐			
	count[2]	☐	☑	☐			
	count[3]	☐	☑	☐			
	count[4]	☐	☑	☐			
	count[5]	☐	☑	☐			
	count[6]	☐	☑	☐			
	count[7]	☐	☑	☐			
	data_out[0..5]	☑	☑	☐			
Storage Qualifier 1	data_out[6]	☑	☑	☑	0	1	
Storage Qualifier 2	data_out[7]	☑	☑	☑	1	0	
	start_sig_pulse	☐	☑	☐			

Signal Configuration:
Clock: clk
Data
Sample depth: 128 RAM type: M4K/M9K
☐ Segmented: 128 1 sample segments
Storage qualifier
Type: Start/Stop
Input port:
☑ Record data discontinuities
☐ Disable storage qualifier
Trigger

Figure 9-17: Data Acquisition of a Recurring Data Pattern in Continuous Mode (Start/Stop)

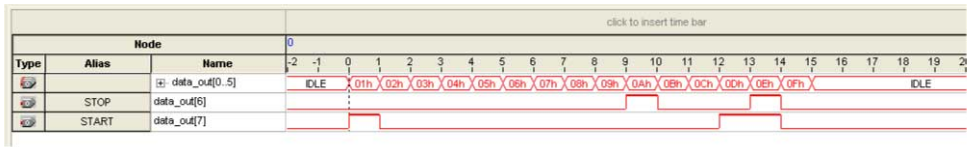
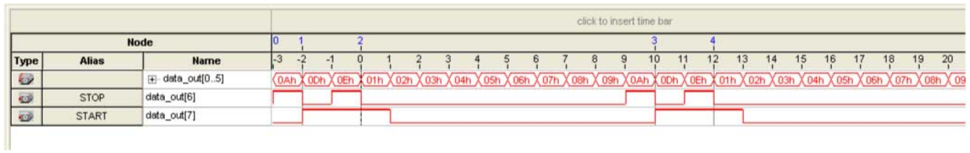


Figure 9-18: Data Acquisition of a Recurring Data Pattern with Start/Stop Storage Qualifier Enabled



State-Based

The State-based storage qualification mode is used with the State-based triggering flow. The state based triggering flow evaluates an if-else based language to define how data is written 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 the storage qualifier feature is enabled for the State-based flow, two additional commands are available, the `start_store` and `stop_store` commands. These commands operate similarly to the Start/Stop capture conditions described in the previous section. Upon the start of acquisition, data is not written 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 are performed within the same clock cycle, a single sample is stored into the acquisition buffer.

Related Information

State-Based Triggering on page 9-32

Showing Data Discontinuities

When you turn on **Record data discontinuities**, the SignalTap II 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.

Disable Storage Qualifier

You can turn off the storage qualifier quickly with the **Disable Storage Qualifier** option, and perform a continuous capture. This option is run-time reconfigurable; that is, the setting can be changed without recompiling the project. Changing storage qualifier mode from the **Type** field requires a recompilation of the project.

Related Information

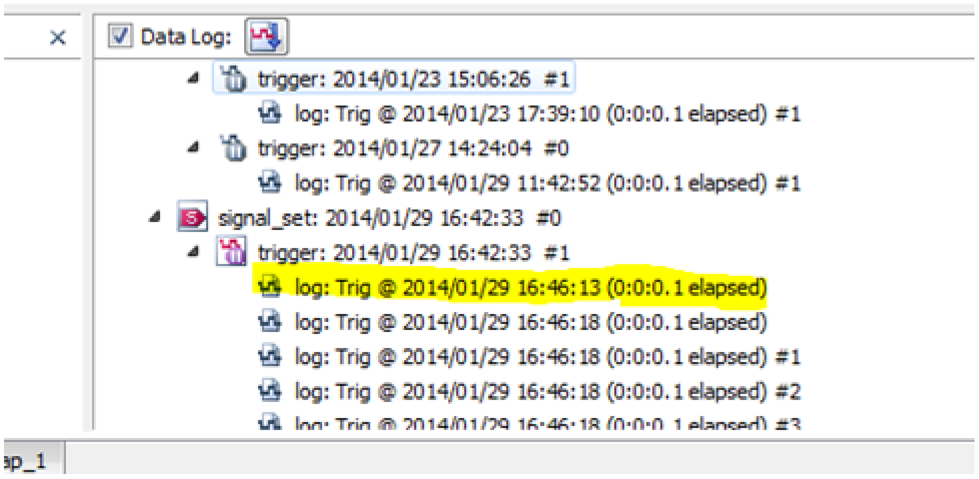
Runtime Reconfigurable Options on page 9-48

Managing Multiple SignalTap II Files and Configurations

You can use more than one **.stp** in one design. Each file potentially has a different group of monitored signals. These signal groups make it possible to debug different blocks in your design. In turn, each group of signals can also be used to define different sets of trigger conditions. Along with each **.stp**, there is also an associated programming file (SRAM Object File [**.sof**]). The settings in a selected SignalTap II file must match the SignalTap II logic design in the associated **.sof** for the logic analyzer to run properly when the device is programmed. Use the Data Log feature and the SOF Manager to manage all of the **.stp** files and their associated settings and programming files.

The Data Log allows you to store multiple SignalTap II configurations within a single **.stp**. Figure 9-19 shows two signal set configurations with multiple trigger conditions in one **.stp**. To toggle between the active configurations, double-click on an entry in the Data Log. As you toggle between the different configurations, the signal list and trigger conditions change in the **Setup** tab of the **.stp**. The active configuration displayed in the **.stp** is indicated by the blue square around the signal specified in the Data Log. Enable the Data Log by clicking the check box next to **Data Log**. To store a configuration in the Data Log, on the Edit menu, click **Save to Data Log** or click the **Save to Data Log** icon at the top of the Data Log. The time stamping for the Data Log entries display the wall-clock time when SignalTap II triggered and the elapsed time from when acquisition started to when the device triggered.

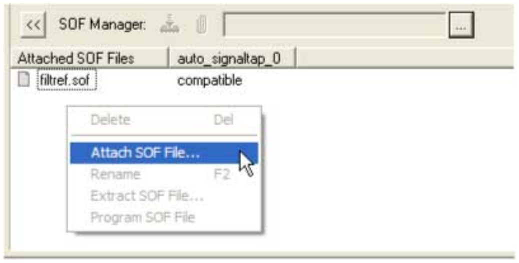
Figure 9-19: Data Log



```
trigger: 2014/01/23 15:06:26  #1
    log: Trig @ 2014/01/23 17:39:10 (0:0:0.1 elapsed) #1
trigger: 2014/01/27 14:24:04  #0
    log: Trig @ 2014/01/29 11:42:52 (0:0:0.1 elapsed) #1
signal_set: 2014/01/29 16:42:33  #0
  trigger: 2014/01/29 16:42:33  #1
    log: Trig @ 2014/01/29 16:46:13 (0:0:0.1 elapsed)
    log: Trig @ 2014/01/29 16:46:18 (0:0:0.1 elapsed)
    log: Trig @ 2014/01/29 16:46:18 (0:0:0.1 elapsed) #1
    log: Trig @ 2014/01/29 16:46:18 (0:0:0.1 elapsed) #2
    log: Trig @ 2014/01/29 16:46:18 (0:0:0.1 elapsed) #3
```

The SOF Manager allows you to embed multiple SOFs into one **.stp**. Embedding an SOF in an **.stp** lets you move the **.stp** to a different location, either on the same computer or across a network, without the need to include the associated **.sof** separately. To embed a new SOF in the **.stp**, right-click in the SOF Manager, and click **Attach SOF File** .

Figure 9-20: SOF Manager



Attached SOF Files	auto_signaltap_0
filtref.sof	compatible

As you switch between configurations in the Data Log, you can extract the SOF that is compatible with that particular configuration. You can use the programmer in the SignalTap II Logic Analyzer to download the new SOF to the FPGA, ensuring that the configuration of your **.stp** always matches the design programmed into the target device.

Define Triggers

When you start the SignalTap II Logic Analyzer, it samples activity continuously from the monitored signals. The SignalTap II Logic Analyzer “triggers”—that is, the logic analyzer stops and displays the data —when a condition or set of conditions that you specified has been reached. This section describes the various types of trigger conditions that you can specify using the SignalTap II Logic Analyzer on the **Signal Configuration** pane.

Creating Basic Trigger Conditions

The simplest kind of trigger condition is a basic trigger. Select this from the list at the top of the **Trigger Conditions** column in the node list in the SignalTap II Logic Analyzer Editor. If you select the **Basic AND** or **Basic OR** trigger type, you must specify the trigger pattern for each signal you have added in the **.stp**. To specify the trigger pattern, right-click in 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 added to the **.stp** 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.

For more information about creating and using mnemonic tables, refer to **View, Analyze, and Use Captured Data**, and to the Quartus Prime Help.

For signals added with certain plug-ins, you can create basic triggers easily using predefined mnemonic table entries. For example, with the Nios II plug-in, if you have specified an **.elf** 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 specified code function name.

Data capture stops and the data is stored in the buffer when the logical `AND` of all the signals for a given trigger condition evaluates to `TRUE`.

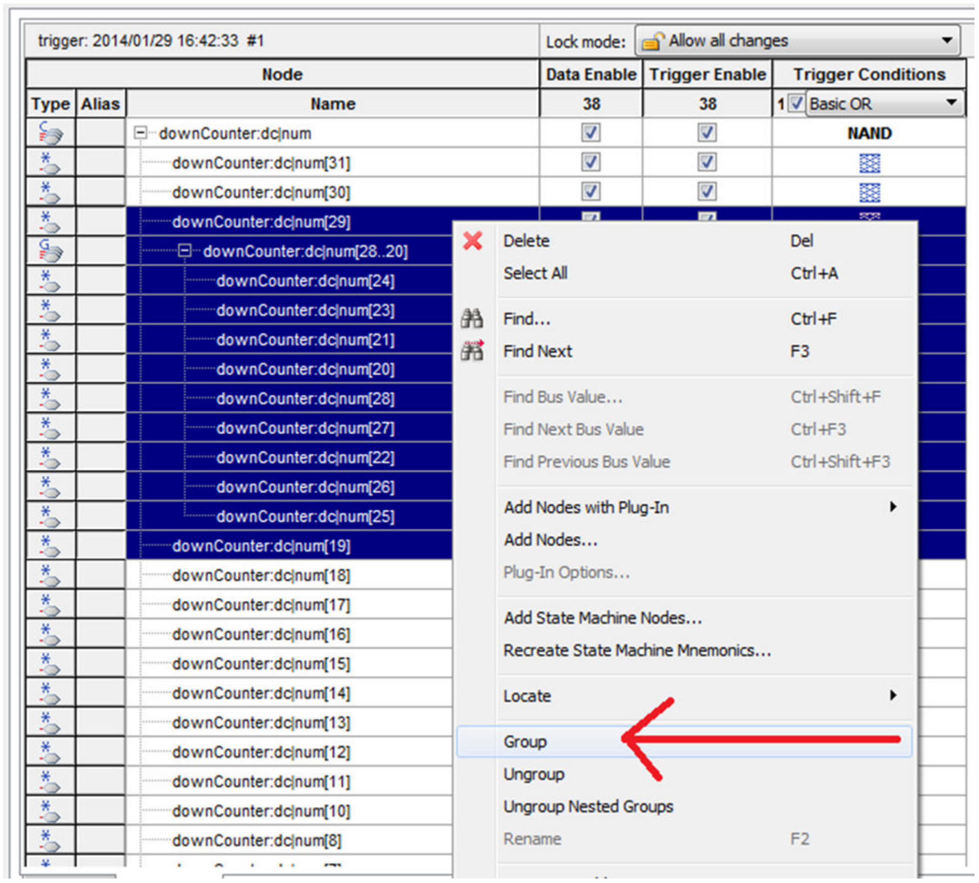
Using the Basic OR Triggering Condition with Nested Groups

When you specify a set of signals as a nested group (group of groups) with the **Basic OR** trigger type, an advanced trigger condition is generated. This advanced trigger 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 generated advanced trigger condition does not change. You can modify the sibling relationships of nodes and not need to recompile your design. The precedence of how this trigger condition is evaluated starts at the bottom-level with the leaf-groups first, then their resulting logic 1 or logic 0 value is used to compute the result of their parent group's logic value. Specifying a value of **TRUE** for a group sets that group's logical result to logic 1 and effectively eliminates all members beneath it from affecting the result of the group trigger. Specifying a value of **FALSE** for a group sets that group's logical result to logic-0 and effectively eliminates all members beneath it from affecting the result of the group trigger.

1. Select **Basic OR** under **Trigger Conditions**.
2. In the **Setup** tab, select nodes including groups.
3. Right-click in the **Setup** tab and select **Group**.
4. Select your signal(s) 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: The OR and AND group trigger conditions are only selectable for groups with no groups as children (bottom-level groups).

Figure 9-21: Creating Nested Groups



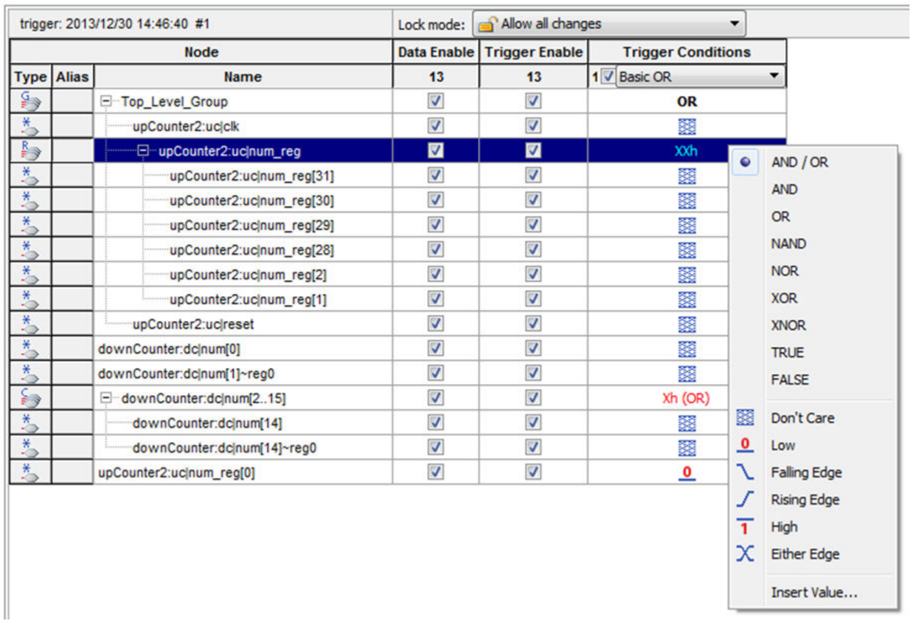
trigger: 2014/01/29 16:42:33 #1

Lock mode: Allow all changes

Type	Alias	Name	Data Enable	Trigger Enable	Trigger Conditions
			38	38	1 ✓ Basic OR
		⊟ downCounter:dc|num	✓	✓	NAND
		downCounter:dc|num[31]	✓	✓	▨
		downCounter:dc|num[30]	✓	✓	▨
		downCounter:dc|num[29]			
		⊟ downCounter:dc|num[28..20]			
		downCounter:dc|num[24]			
		downCounter:dc|num[23]			
		downCounter:dc|num[21]			
		downCounter:dc|num[20]			
		downCounter:dc|num[28]			
		downCounter:dc|num[27]			
		downCounter:dc|num[22]			
		downCounter:dc|num[26]			
		downCounter:dc|num[25]			
		downCounter:dc|num[19]			
		downCounter:dc|num[18]			
		downCounter:dc|num[17]			
		downCounter:dc|num[16]			
		downCounter:dc|num[15]			
		downCounter:dc|num[14]			
		downCounter:dc|num[13]			
		downCounter:dc|num[12]			
		downCounter:dc|num[11]			
		downCounter:dc|num[10]			
		downCounter:dc|num[8]			

Delete Del
Select All Ctrl+A
Find... Ctrl+F
Find Next F3
Find Bus Value... Ctrl+Shift+F
Find Next Bus Value Ctrl+F3
Find Previous Bus Value Ctrl+Shift+F3
Add Nodes with Plug-In
Add Nodes...
Plug-In Options...
Add State Machine Nodes...
Recreate State Machine Mnemonics...
Locate
Group
Ungroup
Ungroup Nested Groups
Rename F2

Figure 9-22: Applying Group Trigger Condition


trigger: 2013/12/30 14:46:40 #1 Lock mode: Allow all changes

Type	Alias	Name	Data Enable 13	Trigger Enable 13	Trigger Conditions 1 ✓ Basic OR
		Top_Level_Group	✓	✓	OR
		upCounter2:uc|clk	✓	✓	▨
		upCounter2:uc|num_reg	✓	✓	XXh
		upCounter2:uc|num_reg[31]	✓	✓	▨
		upCounter2:uc|num_reg[30]	✓	✓	▨
		upCounter2:uc|num_reg[29]	✓	✓	▨
		upCounter2:uc|num_reg[28]	✓	✓	▨
		upCounter2:uc|num_reg[2]	✓	✓	▨
		upCounter2:uc|num_reg[1]	✓	✓	▨
		upCounter2:uc|reset	✓	✓	▨
		downCounter:dc|num[0]	✓	✓	▨
		downCounter:dc|num[1]~reg0	✓	✓	▨
		downCounter:dc|num[2..15]	✓	✓	Xh (OR)
		downCounter:dc|num[14]	✓	✓	▨
		downCounter:dc|num[14]~reg0	✓	✓	▨
		upCounter2:uc|num_reg[0]	✓	✓	0

Creating Advanced Trigger Conditions

With the basic triggering capabilities of the SignalTap II Logic Analyzer, you can build more complex triggers with extra logic that enables you to capture data when a combination of conditions exist. If you select the **Advanced** trigger type at the top of the **Trigger Conditions** column in the node list of the SignalTap II Logic Analyzer Editor, a new tab named **Advanced Trigger** appears where you can build a complex trigger expression using a simple GUI. Drag-and-drop operators into the Advanced Trigger Configuration Editor window to build the complex trigger condition in an expression tree. To configure the operators' settings, double-click or right-click the operators that you have placed and select **Properties**.

Table 9-2: Advanced Triggering Operators

Name of Operator	Type
Less Than	Comparison
Less Than or Equal To	Comparison
Equality	Comparison
Inequality	Comparison
Greater Than	Comparison
Greater Than or Equal To	Comparison
Logical `NOT`	Logical
Logical `AND`	Logical
Logical `OR`	Logical

Name of Operator	Type
Logical XOR	Logical
Reduction AND	Reduction
Reduction OR	Reduction
Reduction XOR	Reduction
Left Shift	Shift
Right Shift	Shift
Bitwise Complement	Bitwise
Bitwise AND	Bitwise
Bitwise OR	Bitwise
Bitwise XOR	Bitwise
Edge and Level Detector	Signal Detection

Note to table :

1. For more information about each of these operators, refer to the Quartus Prime Help.

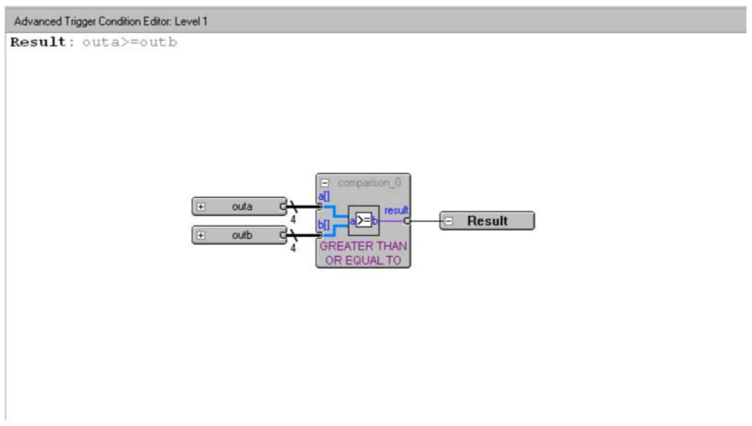
Adding many objects to the Advanced Trigger Condition Editor can make the work space cluttered and difficult to read. To keep objects organized while you build your advanced trigger condition, use the shortcut menu and select **Arrange All Objects**. You can also use the **Zoom-Out** command to fit more objects into the Advanced Trigger Condition Editor window.

Examples of Advanced Triggering Expressions

The following examples show how to use Advanced Triggering:

Figure 9-23: Bus outa Is Greater Than or Equal to Bus outb

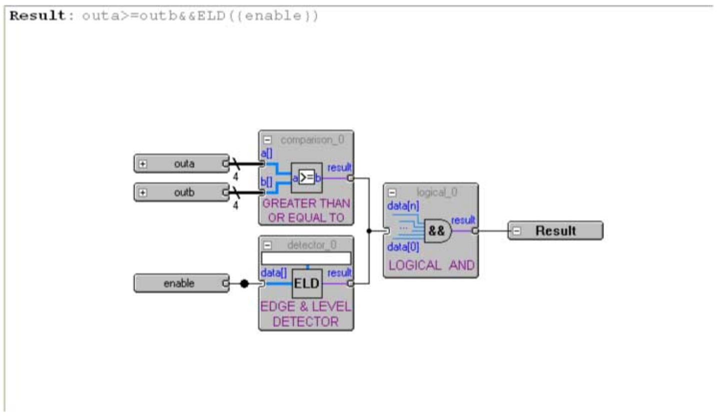
Trigger when bus `outa` is greater than or equal to `outb`.


Advanced Trigger Condition Editor: Level 1

```
Result: outa>=outb
```

Figure 9-24: Enable Signal Has a Rising Edge

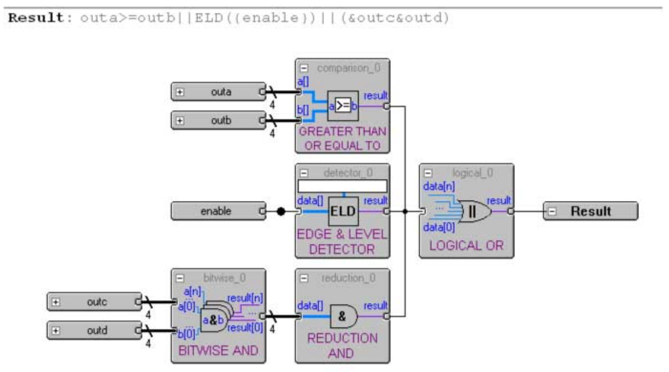
Trigger when bus `outa` is greater than or equal to bus `outb`, and when the enable signal has a rising edge.



Result: outa>=outb&&ELD((enable))

Figure 9-25: 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.

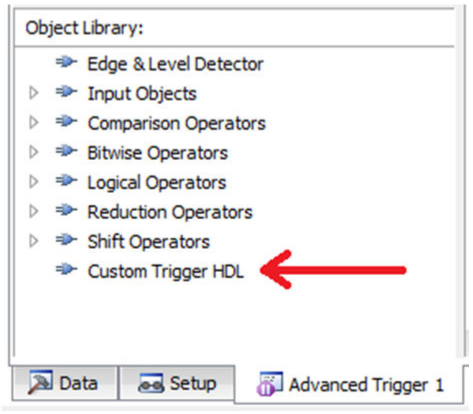


Result: outa>=outb||ELD((enable))||(&outc&outd)

Custom Trigger HDL Object

The Custom Trigger HDL object found in the **Advanced Trigger** editor allows you to create a customized trigger condition with your own HDL module in either Verilog or VHDL. You can use this object to simulate the behavior of your triggering logic to make sure that the logic itself is not faulty. You can tap specific instances of modules located anywhere in the hierarchy of your design without having to manually route all the necessary connections.

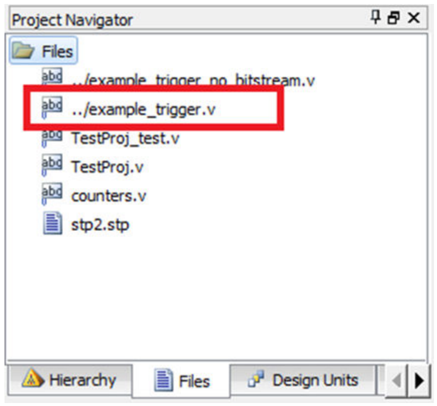
Figure 9-26: Object Library



Custom Trigger Flow

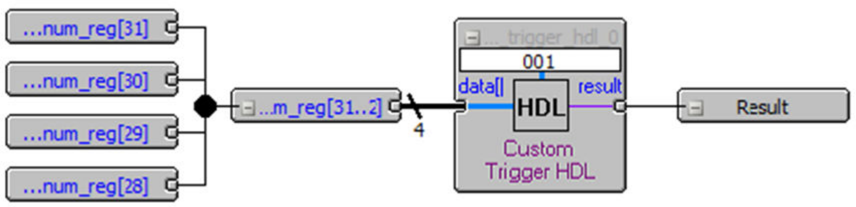
1. Select Advanced for a given trigger-level to make the Advanced Trigger editor active.
2. Prepare your Custom Trigger HDL module. You can either add a new source file to Quartus Prime that contains the trigger module or append the HDL for your trigger module to a source file already included in Quartus Prime **Files** under the **Project Navigator**.

Figure 9-27: Project Navigator



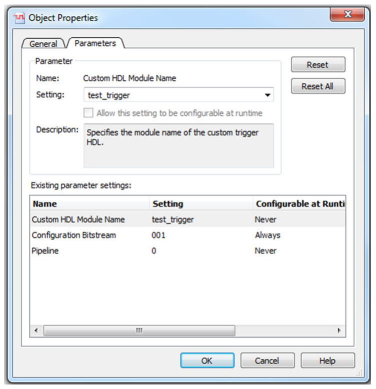
3. Implement the required inputs and outputs for your Custom Trigger HDL module, see **Table 9-3**.
4. Drag in your Custom Trigger HDL object and connect the object's data input bus and result output bit to the final trigger result.

Figure 9-28: Custom Trigger HDL Object



5. Right-click your Custom Trigger HDL object and configure the object's properties, see **Table 9-4**.

Figure 9-29: 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

6. Compile your design.
7. Acquire data with SignalTap II using your custom Trigger HDL object.

Table 9-3: Custom Trigger HDL Module Required Inputs and Outputs

Name	Description	Input/Output	Required/ Optional
`acq_clk`	Acquisition clock used by SignalTap II	Input	Required
`reset`	Reset signal used by SignalTap II when restarting a capture.	Input	Required
`data_in`	• Data input to be connected in the Advanced Trigger editor. • Data your module will use to trigger.	Input	Required
`pattern_in`	• Module's input for the configuration bitstream property. • Runtime configurable property that can be set from SignalTap II GUI to change the behavior of your trigger logic.	Input	Optional
`trigger_out`	Output signal of your module to be asserted when triggering conditions have been met.	Output	Required

Table 9-4: Custom Trigger HDL Module Properties

Property	Description
Custom HDL Module Name	Module name of your triggering logic i.e. module **trigger_foo** (input x, y ...);

Property	Description
Configuration Bitstream	• Allows you to create runtime-configurable trigger logic which can change its behavior based upon the value of the configuration bitstream. • Configuration bitstream property is interpreted as binary and should only contain the characters 1 and 0. The bit-width (number of 1s and 0s) should match the `pattern_in` bit width in **Table 9-3**. • A blank configuration bitstream implies that your module does not have a `pattern_in` input.
Pipeline	• Tells the advanced trigger editor how many stages of pipeline your triggering logic has. • If it takes three clock cycles after a triggering input is received for the trigger output to be asserted, you can denote a pipeline value of three.

Figure 9-30: Example of Verilog Trigger Using 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
```

Figure 9-31: Example of Verilog Trigger with No Configuration Bitstream

```
module test_trigger_no_bs(input acq_clk, reset, input [3:0]
data_in, output trigger_out);

    assign trigger_out = &data_in;

endmodule
```

Trigger Condition Flow Control

The SignalTap II Logic Analyzer offers multiple triggering conditions to give you precise control of the method in which data is captured into the acquisition buffers. Trigger Condition Flow allows you to define the relationship between a set of triggering conditions. The SignalTap II Logic Analyzer **Signal Configuration** pane offers two flow control mechanisms for organizing trigger conditions:

- **Sequential Triggering**—The 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**—Allows you 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.

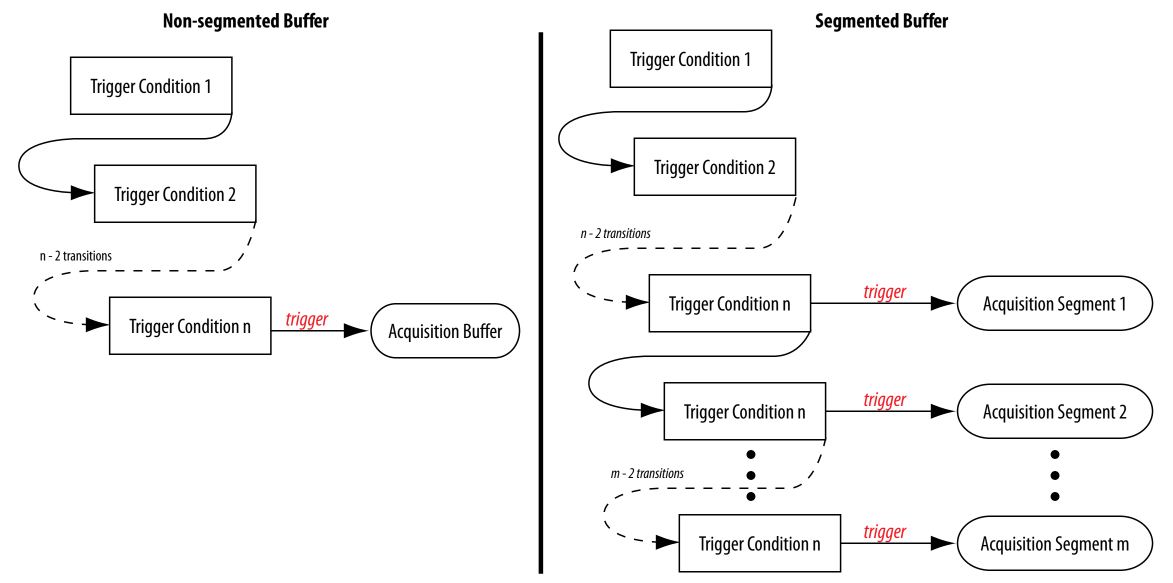
You can use sequential or state based triggering with either a segmented or a non-segmented buffer.

Sequential Triggering

Sequential triggering flow allows you to cascade up to 10 levels of triggering conditions. The SignalTap II Logic Analyzer sequentially evaluates each of the triggering conditions. When the last triggering condition evaluates to `TRUE`, the SignalTap II Logic Analyzer triggers the acquisition buffer. For segmented buffers, every acquisition segment after the first segment triggers on the last triggering condition that you have specified. Use the Simple Sequential Triggering feature with basic triggers, advanced triggers, or a mix of both. Figure 9-32 illustrates the simple sequential triggering flow for non-segmented and segmented buffers.

Note: The external trigger is considered as trigger level `0`. The external trigger must be evaluated before the main trigger levels are evaluated.

Figure 9-32: Sequential Triggering Flow



Notes to figure:

1. The acquisition buffer stops capture when all n triggering levels are satisfied, where $n \leq 10$.
2. An external trigger input, if defined, is evaluated before all other defined trigger conditions are evaluated.

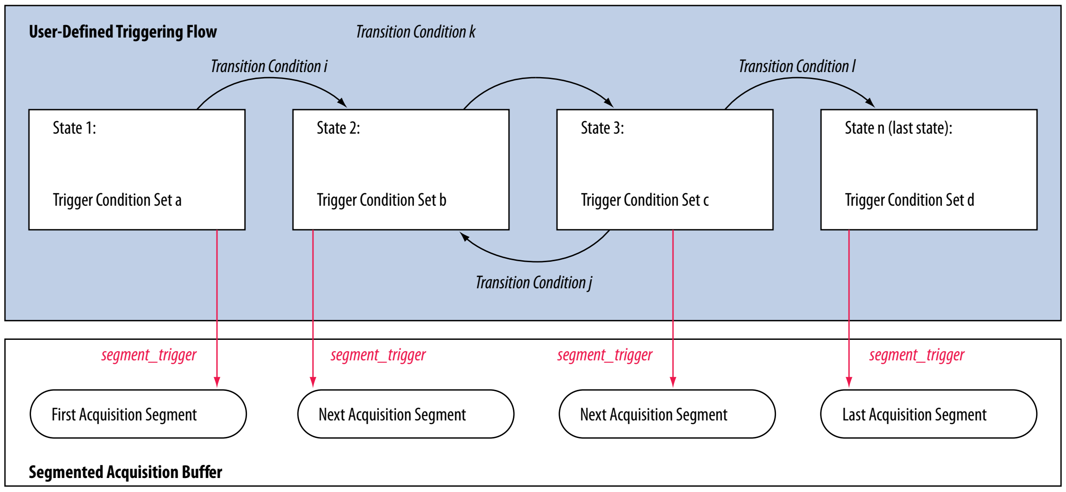
To configure the SignalTap II Logic Analyzer for Sequential triggering, in the SignalTap II editor on the **Trigger flow control** list, select **Sequential**. Select the desired number of trigger conditions from the **Trigger Conditions** list. After you select the desired number of trigger conditions, configure each trigger condition in the node list. To disable any trigger condition, turn on the trigger condition at the top of the column in the node list.

State-Based Triggering

Custom State-based triggering provides the most control over triggering condition arrangement. The State-Based Triggering flow allows you to describe the relationship between triggering conditions precisely, using an intuitive GUI and the SignalTap II Trigger Flow Description Language, a simple description language based upon conditional expressions. Tooltips within the custom triggering flow GUI allow you to describe your desired flow quickly. The custom State-based triggering flow allows for more efficient use of the space available in the acquisition buffer because only specific samples of interest are captured.

Events that trigger the acquisition buffer are organized by a state diagram that you define. All actions performed by the acquisition buffer are captured by the states and all transition conditions between the states are defined by the conditional expressions that you specify within each state.

Figure 9-33: State-Based Triggering Flow



Notes to figure:

1. You are allowed up to 20 different states.
2. An external trigger input, if defined, is evaluated before any conditions in the custom State-based triggering flow are evaluated.

Each state allows you to define a set of conditional expressions. Each conditional expression is a Boolean expression dependent on a combination of triggering conditions (configured within the **Setup** tab), counters, and status flags. Counters and status flags are resources provided by the SignalTap II Logic Analyzer custom-based triggering flow.

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 captured before stopping acquisition of the current segment. The count argument allows you to control the amount of data captured precisely before and after triggering event.

Resource manipulation actions allow you to increment and decrement counters or set and clear status flags. The counter and status flag resources are used as optional inputs in conditional expressions.

Counters and status flags are useful for counting the number of occurrences of particular events and for aiding in triggering flow control.

This SignalTap II custom State-based triggering flow allows you to capture a sequence of events that may not necessarily be contiguous in time; for example, capturing a communication transaction between two devices that includes a handshaking protocol containing a sequence of acknowledgements.

The **State-Based Trigger Flow** tab is the control interface for the custom state-based triggering flow. To enable this tab, select **State-based** on the **Trigger Flow Control** list. (Note that when **Trigger Flow Control** is specified as **Sequential**, the **State-Based Trigger Flow** tab is hidden.)

The **State-Based Trigger Flow** tab is partitioned into the following three panes:

- **State Diagram** pane
- **Resources** pane
- **State Machine** pane

State Diagram Pane

The **State Diagram** pane provides a graphical overview of the triggering flow that you define. It shows the number of states available and the state transitions between the states. You can adjust the number of available states by using the menu above the graphical overview.

State Machine Pane

The **State Machine** pane contains the text entry boxes where you can define the triggering flow and actions associated with each state. You can define the triggering flow using the SignalTap II 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 Information

SignalTap II Trigger Flow Description Language on page 9-34

Resources Pane

The **Resources** pane allows you to declare Status Flags and Counters for use in the conditional expressions in the Custom Triggering Flow. Actions to decrement and increment counters or to set and clear status flags are performed within the triggering flow that you define.

You can specify up to 20 counters and 20 status flags. Counter and status flags values may be initialized by right-clicking the status flag or counter name after selecting a number of them from the respective pull-down list, and selecting **Set Initial Value.** To specify a counter width, right-click the counter name and select **Set Width**. Counters and flag values are updated dynamically after acquisition has started to assist in debugging your trigger flow specification.

The **configurable at runtime** options in the **Resources** pane allows you to configure the custom-flow control options that can be changed at runtime without requiring a recompilation.

Table 9-5: 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.

You can restrict changes to your SignalTap II configuration to include only the options that do not require a recompilation. Trigger lock-mode allows you to make changes that can be immediately reflected in the device.

1. On the **Setup** tab, select **Allow trigger condition changes only**.
2. Modify the Trigger Flow conditions in the **Custom Trigger Flow** tab.
3. Click the desired parameter in the text box and select a new parameter from the menu that appears.

Note: Trigger lock mode restricts changes to the configuration settings that have **configurable at runtime** specified. The runtime configurable settings for the **Custom Trigger Flow** tab are on by default. You may get some performance advantages by disabling some of the runtime configurable options.

Related Information

- Performance and Resource Considerations on page 9-45
- Runtime Reconfigurable Options on page 9-48

SignalTap II 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. Each line in the example shows a language format. Keywords are shown in bold. Non-terminals are delimited by "<>" and are further explained in the following sections. Optional arguments are delimited by "`[]`".

```
state <State_label>:
<action_list>

if( <Boolean_expression> )
<action_list>
[else if ( <boolean_expression> )
<action_list>]
[else
<action_list>]
```

Note: Multiple `else if` conditions are allowed.

The priority for evaluation of conditional statements is assigned from top to bottom. The *<boolean_expression>* in an `if` statement can contain a single event, or it can contain multiple event conditions. The `action_list` within an `if` or an `else if` clause must be delimited by the begin and end tokens when the action list contains multiple statements. When the boolean expression is evaluated `TRUE`, the logic analyzer analyzes all of the commands in the action list concurrently. The possible actions include:

- Triggering the acquisition buffer
- Manipulating a counter or status flag resource
- Defining a state transition

Related Information

Custom Triggering Flow Application Examples on page 9-61

State Labels

State labels are identifiers that can be used in the action `goto`.

`state` <*state_label*>: begins the description of the actions evaluated when this state is reached.

The description of a state ends with the beginning of another state or the end of the whole trigger flow description.

Boolean_expression

`Boolean_expression` is a collection of logical operators, relational operators, and their operands that evaluate into a Boolean result. Depending on the operator, the operand can be a reference to a trigger condition, a counter and a register, or a numeric value. Within an expression, parentheses can be used to group a set of operands.

Logical operators accept any boolean expression as an operand.

Table 9-6: Logical Operators

Operator	Description	Syntax
`!`	`NOT` operator	`! expr1`
`&&`	`AND` operator	`expr1 && expr2`
`||`	`OR` operator	`expr1 || expr2`

Relational operators are performed on counters or status flags. The comparison value, the right operator, must be a numerical value.

Table 9-7: Relational Operators

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>`

Operator	Description	Syntax
<=	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.

Action_list

Action_list is a list of actions that can be performed when a state is reached and a condition is also satisfied. If more than one action is specified, they must be enclosed by `begin` and `end`. The actions can be categorized as resource manipulation actions, buffer control actions, and state transition actions. Each action is terminated by a semicolon (`;`).

Resource Manipulation Action

The resources used in the trigger flow description can be either counters or status flags.

Table 9-8: Resource Manipulation Action

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 Action

Buffer control actions specify an action to control the acquisition buffer.

Table 9-9: Buffer Control Action

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>;`

Action	Description	Syntax
`segment_trigger`	Ends the acquisition of the current segment. The SignalTap II Logic Analyzer starts acquiring from the next segment on evaluating this command. If all segments are filled, the oldest segment is overwritten with the latest sample. The acquisition stops when a trigger action is evaluated. This action cannot be used in non-segmented acquisition mode.	`segment_trigger <post-fill_count>;`
`start_store`	Asserts the `write_enable` to the SignalTap II acquisition buffer. This command is active only when the State-based storage qualifier mode is enabled.	`start_store`
`stop_store`	De-asserts the `write_enable` signal to the SignalTap II acquisition buffer. This command is active only when the State-based storage qualifier mode is enabled.	`stop_store`

Both `trigger` and `segment_trigger` actions accept an optional post-fill count argument. If provided, the current acquisition acquires the number of samples provided by post-fill count and then stops acquisition. If no post-count value is specified, the trigger position for the affected buffer defaults to the trigger position specified in the **Setup** tab.

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 whether or not the post-fill count has been satisfied for the current buffer. The remaining unfilled post-count acquisitions in the current buffer are discarded and displayed as grayed-out samples in the data window.

State Transition Action

The State Transition action specifies the next state in the custom state control flow. It is specified by the `goto` command. The syntax is as follows:

```
goto <state_label>;
```

Using the State-Based Storage Qualifier Feature

When you select State-based for the storage qualifier type, the `start_store` and `stop_store` actions are enabled in the State-based trigger flow. These commands, when used in conjunction with the expressions of the State-based trigger flow, give you maximum flexibility to control data written into the acquisition buffer.

Note: The `start_store` and `stop_store` commands can only be applied to a non-segmented buffer.

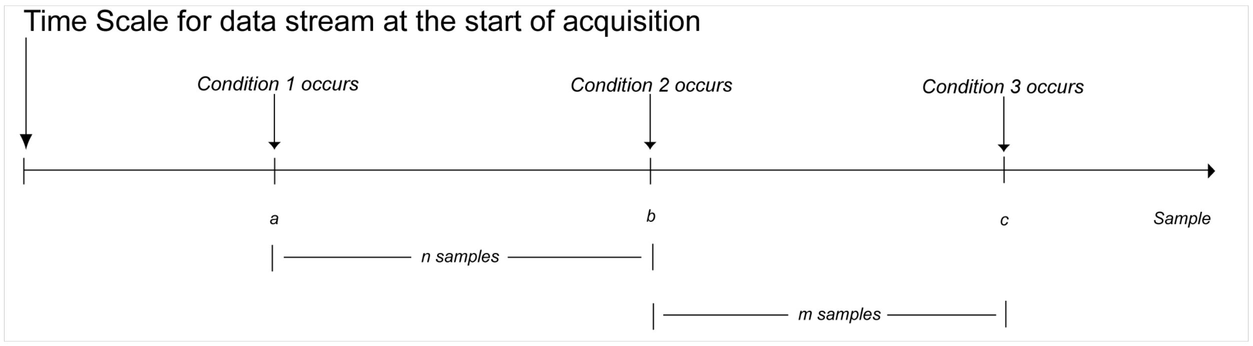
The `start_store` and `stop_store` commands function similar to the start and stop conditions when using the **start/stop** storage qualifier mode conditions. If storage qualification is enabled, the `start_store` command must be issued for SignalTap II to write data into the acquisition buffer. No data is acquired until the `start_store` command is performed. Also, a `trigger` command must be included as part of the trigger flow description. The `trigger` command is necessary to complete the acquisition and display the results on the waveform display.

The following example illustrates the behavior of the State-based trigger flow with the storage qualification commands.

```
State 1: ST1:
if ( condition1 )
   start_store;
else if ( condition2 )
   trigger value;
else if ( condition3 )
   stop_store;
```

Figure 9-34 shows a hypothetical scenario with three trigger conditions that happen at different times after you click **Start Analysis**. The trigger flow description in the example above , when applied to the scenario shown in Figure 9-34, illustrates the functionality of the storage qualification feature for the state-based trigger flow.

Figure 9-34: Capture Scenario for Storage Qualification with the State-Based Trigger Flow



In this example, the SignalTap II Logic Analyzer does not write into the acquisition buffer until sample a, when Condition 1 occurs. Once sample b is reached, the `trigger value` command is evaluated. The logic analyzer continues to write into the buffer to finish the acquisition. The trigger flow specifies a `stop_store` command at sample c, m samples after the trigger point occurs.

The logic analyzer finishes the acquisition and displays the contents of the waveform if it can successfully finish the post-fill acquisition samples before Condition 3 occurs. In this specific case, the capture ends if the post-fill count value is less than *m*.

If the post-fill count value specified in Trigger Flow description 1 is greater than 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 SignalTap II Logic Analyzer continues to evaluate the **stop_store** and **start_store** commands even after the **trigger** command is evaluated. If the acquisition has paused, you can 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 are updated in real-time during a data acquisition.

Figure 9-35 and Figure 9-36 show a real data acquisition of the scenario. Figure 9-35 illustrates a scenario where the data capture finishes successfully. It uses a buffer with a sample depth of 64, $m = n = 10$, and the post-fill count value = 5. Figure 9-36 illustrates a scenario where 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.

Figure 9-35: Storage Qualification with Post-Fill Count Value Less than *m* (Acquisition Successfully Completes)

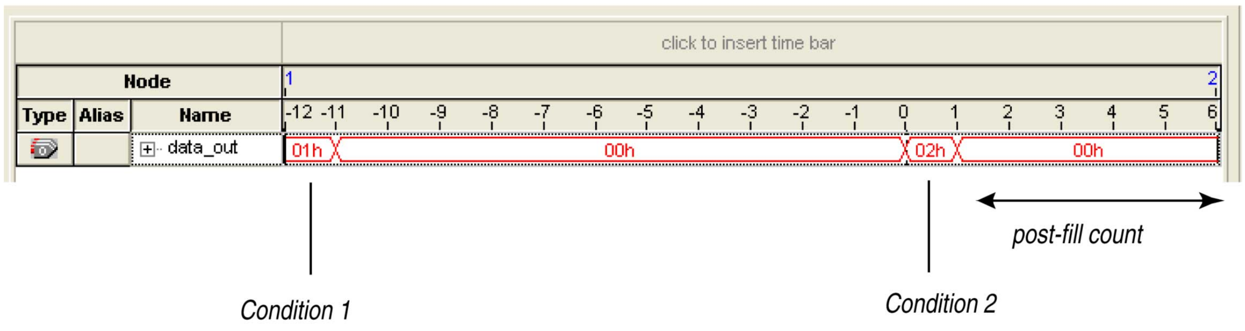
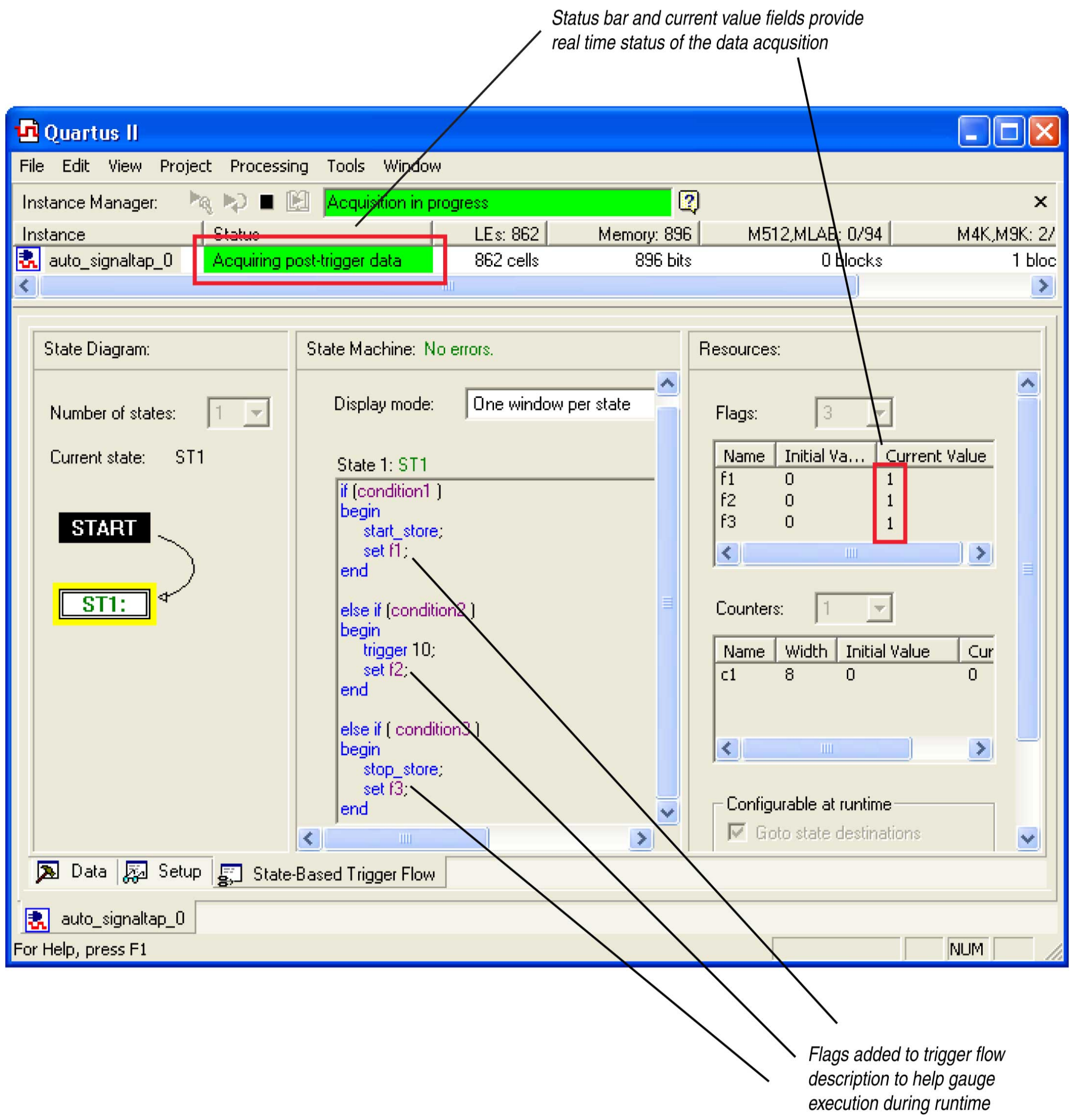


Figure 9-36: Storage Qualification with Post-Fill Count Value Greater than *m* (Acquisition Indefinitely Paused)

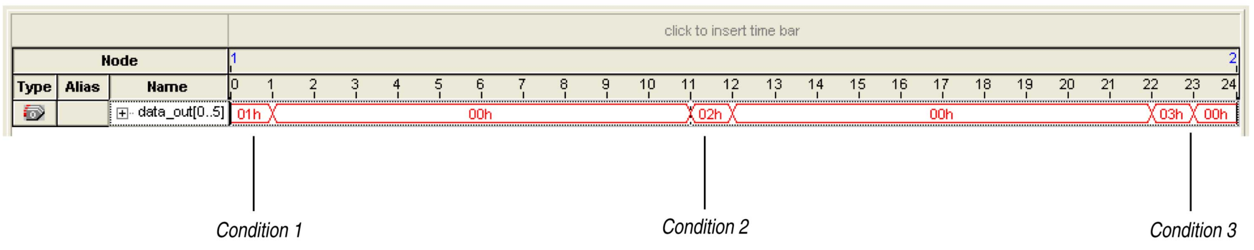


```
if (condition1 )
begin
    start_store;
    set f1;
end

else if (condition2 )
begin
    trigger 10;
    set f2;
end

else if ( condition3 )
begin
    stop_store;
    set f3;
end
```

Name	Initial Va...	Current Value
f1	0	1
f2	0	1
f3	0	1

Name	Width	Initial Value	Cur
c1	8	0	0

Figure 9-37: 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. The code example below shows a trigger flow description that skips three clock cycles of samples after hitting condition 1. **Figure 9-38** shows the data transaction on a continuous capture and **Figure 9-40** shows the data capture with the Trigger flow description applied, in the example below.

```
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
```

Figure 9-38: Continuous Capture of Data Transaction for Example 2

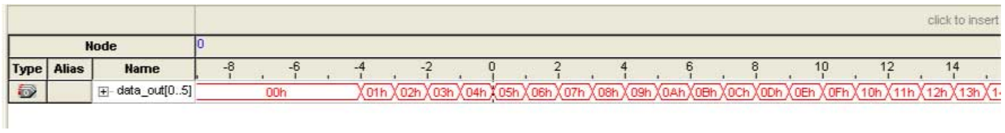
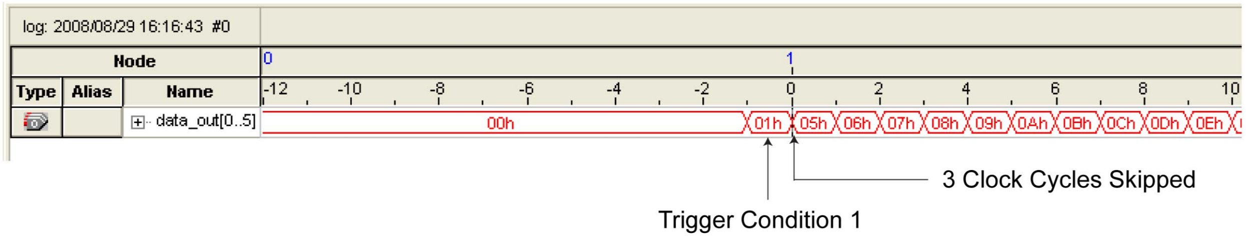

click to insert
Node
Type
Alias
Name
data_out[0..5]
0
-8 -6 -4 -2 0 2 4 6 8 10 12 14
00h 01h 02h 03h 04h 05h 06h 07h 08h 09h 0Ah 0Bh 0Ch 0Dh 0Eh 0Fh 10h 11h 12h 13h

Figure 9-39: Capture of Data Transaction with Trigger Flow Description Applied



Specifying the Trigger Position

The SignalTap II Logic Analyzer allows you to specify the amount of data that is acquired before and after a trigger event. You can specify the trigger position independently between a Runtime and Power-Up Trigger. Select the desired ratio of pre-trigger data to post-trigger data by choosing one of the following ratios:

- **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.

If you use the custom-state based triggering flow, you can specify a custom trigger position. The `segment_trigger` and trigger actions accept a post-fill count argument. The post-fill count specifies the number of samples to capture before stopping data acquisition for the non-segmented buffer or a data segment when using the `trigger` and `segment_trigger` commands, respectively. When the captured data is displayed in the SignalTap II data window, 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.

For segmented buffers, the acquisition segments that have a post-count argument define use of the post-count setting. Segments that do not have a post-count setting default to the trigger position ratios defined in the **Setup** tab.

Related Information

State-Based Triggering on page 9-32

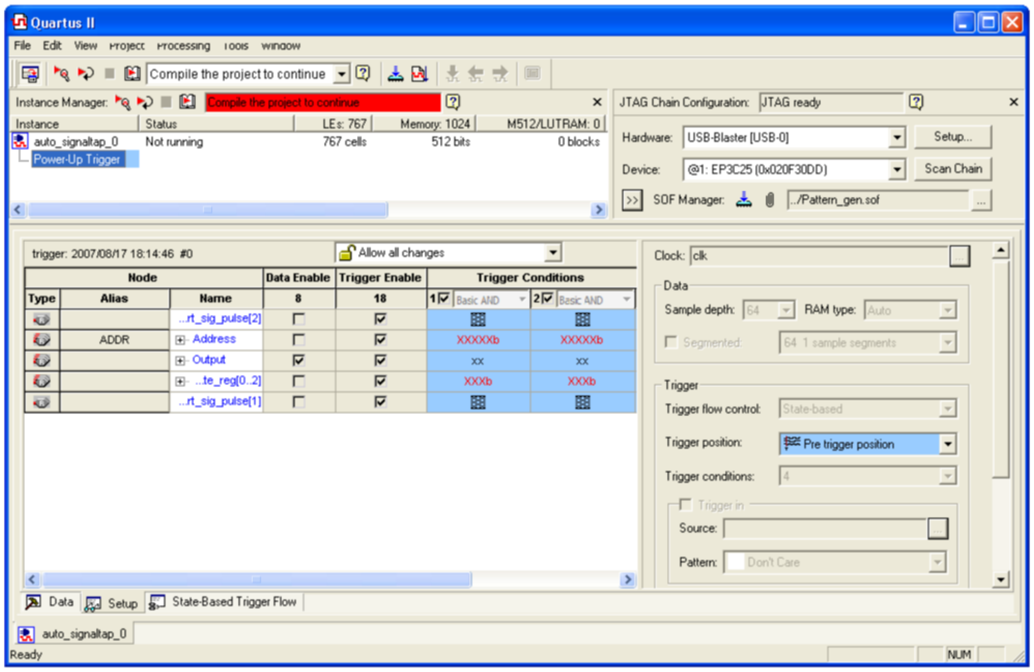
Creating a Power-Up Trigger

Typically, the SignalTap II Logic Analyzer is used to trigger on 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. However, there may be cases when you would like to capture trigger events that occur during device initialization, immediately after the FPGA is powered on or reset. With the SignalTap II Power-Up Trigger feature, you arm the SignalTap II Logic Analyzer and capture data immediately after device programming.

Enabling a Power-Up Trigger

You can add a different Power-Up Trigger to each logic analyzer instance in the **SignalTap II Instance Manager** pane. To enable the Power-Up Trigger for a logic analyzer instance, right-click the instance and click **Enable Power-Up Trigger**, or select the instance, and on the **Edit** menu, click **Enable Power-Up Trigger**. To disable a Power-Up Trigger, click **Disable Power-Up Trigger** in the same locations. Power-Up Trigger is shown as a child instance below the name of the selected instance with the default trigger conditions specified in the node list.

Figure 9-40: SignalTap II Logic Analyzer Editor with Power-Up Trigger Enabled


Quartus II
File Edit View Project Processing Tools Window
Compile the project to continue
Instance Manager:
Compile the project to continue
JTAG Chain Configuration: JTAG ready
Setup...
Scan Chain
SOF Manager:
Data
Setup
State-Based Trigger Flow
auto_signaltap_0
Ready
NUM

Instance	Status	LEs: 767	Memory: 1024	M512/LUTRAM: 0
auto_signaltap_0	Not running	767 cells	512 bits	0 blocks
Power-Up Trigger				

Hardware: USB-Blaster [USB-0]
Device: @1: EP3C25 (0x020F30DD)
SOF Manager: ../Pattern_gen.sof

trigger: 2007/08/17 18:14:46 #0 — Allow all changes

Node			Data Enable	Trigger Enable	Trigger Conditions	
Type	Alias	Name	8	18	1 ✓ Basic AND	2 ✓ Basic AND
		...rt_sig_pulse[2]	☐	✓	▦	▦
	ADDR	Address	☐	✓	XXXXXb	XXXXXb
		Output	✓	✓	xx	xx
		...te_reg[0..2]	☐	✓	XXXb	XXXb
		...rt_sig_pulse[1]	☐	✓	▦	▦

Clock: clk
Data
Sample depth: 64 RAM type: Auto
☐ Segmented: 64 1 sample segments
Trigger
Trigger flow control: State-based
Trigger position: Pre trigger position
Trigger conditions: 4
☐ Trigger in
Source:
Pattern: Don't Care

Managing and Configuring Power-Up and Runtime Trigger Conditions

When the Power-Up Trigger is enabled for a logic analyzer instance, you can create basic and advanced trigger conditions for the trigger as you do with a Run-Time Trigger. Power-Up Trigger conditions that you can adjust are color coded light blue, while Run-Time Trigger conditions you cannot adjust remain white. 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. 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 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 SignalTap II Logic Analyzer instance, even if a similar change to the Runtime Trigger conditions does not require a recompilation.

While creating or making changes to the trigger conditions for the Run-Time Trigger or the Power-Up Trigger, you may want to copy these conditions to the other trigger. This enables you to look for the same trigger during both power-up and runtime. To do this, right-click the instance name or the Power-Up Trigger name in the **Instance Manager** and click **Duplicate Trigger**, or select the instance name or the Power-Up Trigger name and on the **Edit** menu, click **Duplicate Trigger**.

You can also use In-System Sources and Probes in conjunction with the SignalTap II 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 Information

- **Design Debugging Using In-System Sources and Probes documentation** on page 13-1

Using External Triggers

You can create a trigger input that allows you to trigger the SignalTap II Logic Analyzer from an external source. The external trigger input behaves like trigger condition 1, is evaluated, and must be TRUE before any other configured trigger conditions are evaluated. The logic analyzer supplies a signal to trigger external devices or other SignalTap II 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 SignalTap II 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 Altera SoC.

Related Information

- **FPGA-Adaptive Software Debug and Performance Analysis white paper**
  Information about the ARM DS-5 debugging solution
- **Signal Configuration Pane online help**
  Information about setting up external triggers

Compile the Design

When you add an **.stp** to your project, the SignalTap II Logic Analyzer becomes part of your design. You must compile your project to incorporate the SignalTap II logic and enable the JTAG connection you use to control the logic analyzer. When you are debugging with a traditional external logic analyzer, you must often make changes to the signals monitored as well as the trigger conditions.

Preventing Changes Requiring Recompilation

You can 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 to allow only trigger condition changes.

Related Information

Setup Tab (SignalTap II Logic Analyzer) online help

Timing Preservation with the SignalTap II Logic Analyzer

In addition to verifying functionality, timing closure is one of the most crucial processes in successful operation of your design. The Quartus Prime Pro Edition software supports timing preservation for post-fit taps in Arria 10 designs with the Rapid Recompile feature. Rapid Recompile automatically reuses verified portions of your design during recompilations, rather than reprocessing those portions.

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 Information

Timing Closure and Optimization documentation

Performance and Resource Considerations

There is a necessary trade-off between the runtime flexibility of the SignalTap II Logic Analyzer, the timing performance of the SignalTap II Logic Analyzer, and resource usage. The SignalTap II Logic Analyzer allows you to select the runtime configurable parameters to balance the need for runtime flexibility, speed, and area. The default values have been chosen to 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 optimal configuration for your design.

The following tips provide extra timing slack if you have determined that the SignalTap II logic is in your critical path, or to alleviate the resource requirements that the SignalTap II Logic Analyzer consumes if your design is resource-constrained.

If SignalTap II logic is part of your critical path, follow these tips to speed up the performance of the SignalTap II 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 SignalTap II 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**—All signals that you add to the **.stp** have **Trigger Enable** turned on. Turn off **Trigger Enable** for signals that you do not plan to use as triggers.
- **Turn on Physical Synthesis for register retiming**—If you have a large number of triggering signals enabled (greater than the number of inputs that would fit in a LAB) that fan-in logic to a gate-based triggering condition, such as a 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.

If your design is resource constrained, follow these tips to reduce the amount of logic or memory used by the SignalTap II Logic Analyzer:

- **Disable runtime configurable options**—Disabling runtime configurability for advanced trigger conditions or runtime configurable options in the State-based triggering flow results in using fewer LEs.
- **Minimize the number of segments in the acquisition buffer**—You can reduce the number of logic resources used for the SignalTap II Logic Analyzer by limiting the number of segments in your sampling buffer to only those required.
- **Disable the Data Enable for signals that are used for triggering only**—By default, both the **data enable** and **trigger enable** options are selected for all signals. Turning off the **data enable** option for signals used as trigger inputs only saves on memory resources used by the SignalTap II Logic Analyzer.

Because performance results are design-dependent, try these options in different combinations until you achieve the desired balance between functionality, performance, and utilization.

Program the Target Device or Devices

After you compile your project, including the SignalTap II Logic Analyzer, configure the FPGA target device. When you are using the SignalTap II Logic Analyzer for debugging, configure the device from the **.stp** instead of the Quartus Prime Programmer. Because you configure from the **.stp**, you can open more than one **.stp** and program multiple devices to debug multiple designs simultaneously.

The settings in an **.stp** must be compatible with the programming **.sof** used to program the device. An **.stp** is considered compatible with an **.sof** when the settings for the logic analyzer, such as the size of the capture buffer and the signals selected for monitoring or triggering, match the way the target device is programmed. If the files are not compatible, you can still program the device, but you cannot run or control the logic analyzer from the SignalTap II Logic Analyzer Editor.

Note: When the SignalTap II Logic Analyzer detects incompatibility after analysis is started, a system error message is generated containing two CRC values, the expected value and the value retrieved from the **.stp** instance on the device. The CRC values are calculated based on all SignalTap II settings that affect the compilation.

To ensure programming compatibility, make sure to program your device with the latest **.sof** created from the most recent compilation. Checking whether or not a particular **.sof** is compatible with the current SignalTap II configuration is achieved quickly by attaching the **.sof** to the SOF manager.

Before starting a debugging session, do not make any changes to the **.stp** settings that would require recompiling the project. You can check the SignalTap II status display at the top of the **Instance Manager** pane to verify whether a change you made requires recompiling the project, producing a new **.sof**. This feature gives you the opportunity to undo the change, so that you do not need to recompile your project. To prevent these types of changes, select **Allow trigger condition changes only** to lock the **.stp**. The Incremental Route lock mode, **Allow incremental route changes only**, limits changes that require an incremental route using Rapid Recompile, and not a full compile.

Although having a Quartus Prime project is not required when using an **.stp**, it is recommended. The project database contains information about the integrity of the current SignalTap II Logic Analyzer session. Without the project database, there is no way to verify that the current **.stp** matches the **.sof** that is downloaded to the device. If you have an **.stp** that does not match the **.sof**, incorrect data is captured in the SignalTap II Logic Analyzer.

Related Information

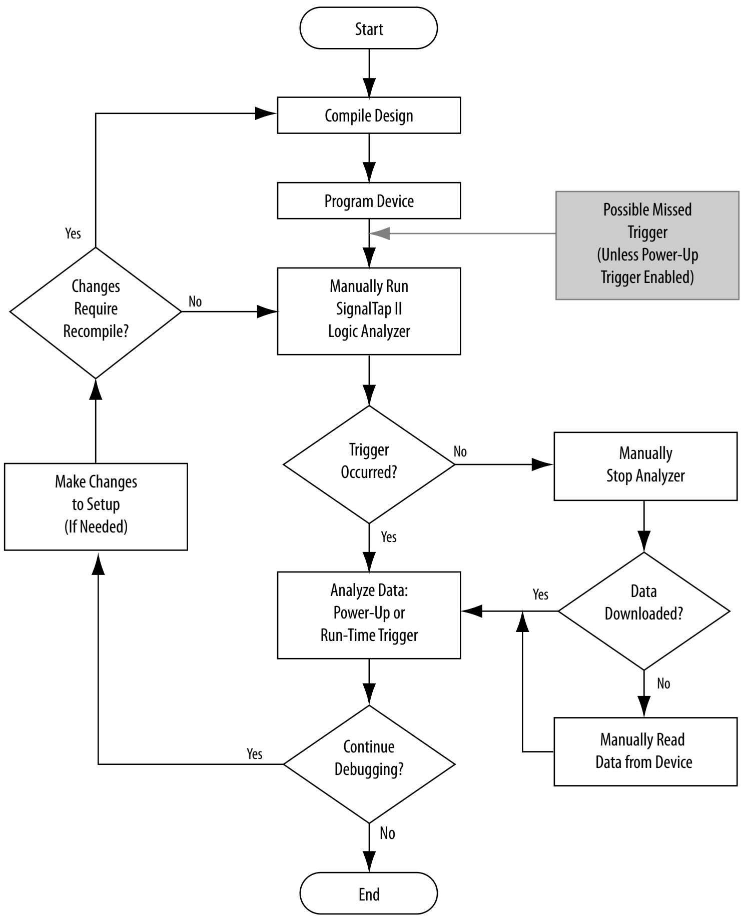
Running the SignalTap II Logic Analyzer online help

Run the SignalTap II Logic Analyzer

After the device is configured with your design that includes the SignalTap II Logic Analyzer, perform debugging operations in a manner similar to when you use an external logic analyzer. You initialize the logic analyzer by starting an analysis. When your trigger event occurs, the captured data is stored in the memory buffer on the device and then transferred to the **.stp** 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. Figure 9-41 illustrates a flow that shows how you operate the SignalTap II Logic Analyzer. The flowchart indicates where Power-Up and Runtime Trigger events occur and when captured data from these events is available for analysis.

Figure 9-41: 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 SignalTap II Logic Analyzer
Trigger Occurred?
No
Manually Stop Analyzer
Make Changes to Setup (If Needed)
Yes
Analyze Data: Power-Up or Run-Time Trigger
Yes
Data Downloaded?
No
Manually Read Data from Device
Yes
Continue Debugging?
No
End

You can also use In-System Sources and Probes in conjunction with the SignalTap II 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 Information

- Design Debugging Using In-System Sources and Probes documentation on page 13-1

Runtime Reconfigurable Options

Certain settings in the **.stp** are changeable without recompiling your design when you use Runtime Trigger mode.

Table 9-10: Runtime Reconfigurable Features

Runtime Reconfigurable Setting	Description
Basic Trigger Conditions and Basic Storage Qualifier Conditions	All signals that have the Trigger condition turned on can be changed to any basic trigger condition value 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 are shown 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	Table 9-5 lists Reconfigurable State-based trigger flow options.

Runtime Reconfigurable options can potentially 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.

You can configure the **.stp** to prevent changes that normally require recompilation. To do this, in the **Setup** tab, select **Allow Trigger Condition changes only** above the node list.

Incremental Route lock mode, **Allow incremental route changes only**, limits changes which will only require an Incremental Route compilation, and not a full compile.

The example below illustrates a potential use case for Runtime Reconfigurable features. This example provides a storage qualified enabled State-based trigger flow description and shows how you can 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
```

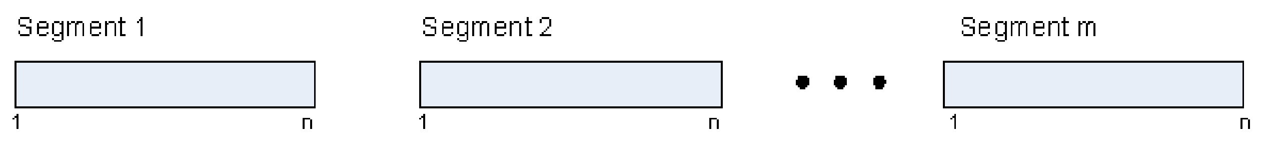
Note: *m* x *n* must equal the sample depth to efficiently use the space in the sample buffer.

Figure 9-42 shows a segmented buffer described by the trigger flow in example above.

During runtime, the values m and n are runtime reconfigurable. By changing the m and n values in the preceding trigger flow description, you can dynamically adjust the segment boundaries without incurring a recompile.

Figure 9-42: Segmented Buffer Created with Storage Qualifier and State-Based Trigger

Total sample depth is fixed, where *m* x *n* must equal sample depth.



You can add states into the trigger flow description and selectively mask out specific states and enable other ones at runtime with status flags.

The example below shows a modified description of the example above 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
```

SignalTap II Status Messages

Table 9-11 describes the text messages that might appear in the SignalTap II 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 9-11: Text Messages in the SignalTap II Status Indicator

Message	Message Description
`Not running`	The SignalTap II 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 SignalTap II Logic Analyzer is performing a Runtime or Power-Up Trigger acquisition and is waiting for the clock signal to transition.
`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 SignalTap II 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 SignalTap II Logic Analyzer is now waiting for the trigger event to occur.
`Trigger level <x>` met	The condition of trigger condition *x* has occurred. The SignalTap II 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 SignalTap II 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 Quartus Prime software through the JTAG chain.
`Ready to acquire`	The SignalTap II Logic Analyzer is waiting for you to initialize the analyzer.

Message	Message Description
Note to Table 9-11 : 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.

View, Analyze, and Use Captured Data

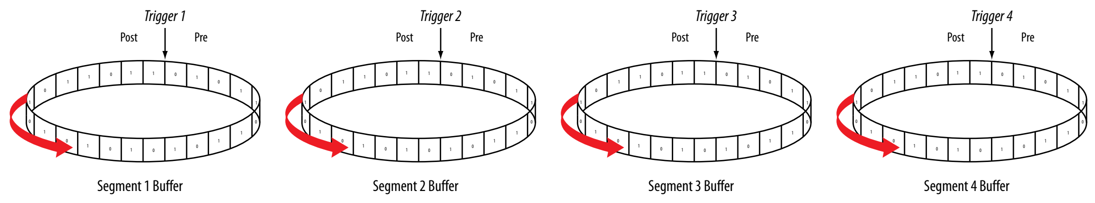
Once a trigger event has occurred or you capture data manually, you can use the SignalTap II interface to examine the data, and use your findings to help debug your design.

When in the Data view, you can use the drag-to-zoom feature by left-clicking to isolate the data of interest.

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 SignalTap II Logic Analyzer performs back-to-back data captures for each acquisition segment within your data buffer. The trigger flow, or the type and order in which the trigger conditions evaluate for each buffer, is defined by either the Sequential trigger flow control or the Custom State-based trigger flow control. Figure 9-43 shows a segmented acquisition buffer with four segments represented as four separate non-segmented buffers.

Figure 9-43: Segmented Acquisition Buffer



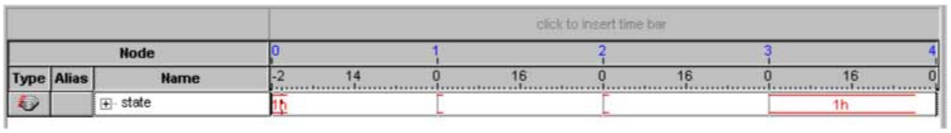
The SignalTap II Logic Analyzer finishes an acquisition with a segment, and advances to the next segment to start a new acquisition. Depending on when a trigger condition occurs, it may affect the way the data capture appears in the waveform viewer. Figure 9-43 illustrates the method in which data is captured. The Trigger markers in Figure 9-43—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 SignalTap II 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 SignalTap II Logic Analyzer to accurately capture all of the trigger conditions that have occurred. Samples that have not been used appear as a blank space in the waveform viewer.

Figure 9-44 shows an example of a capture using sequential flow control with the trigger condition for each segment specified as **Don't Care**. 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 SignalTap II Logic Analyzer allocated to the buffer.

Figure 9-44: Segmented Capture with Preemption of Acquisition Segments

A segmented acquisition buffer using the sequential trigger flow with a trigger condition specified as Don't Care. All segments, with the exception of the last segment, capture only one sample because the next trigger condition preempts the current buffer from filling to completion.



For the sequential trigger flow, the **Trigger Position** option applies to every segment in the buffer. For maximum flexibility on how the trigger position is defined, 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.

Differences in Pre-fill Write Behavior Between Different Acquisition Modes

The SignalTap II 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 SignalTap II 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.

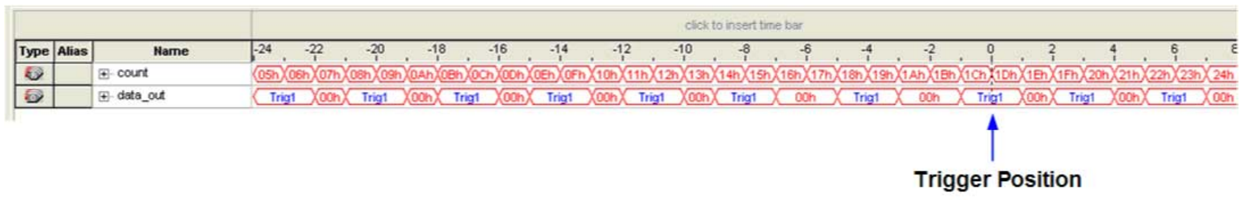
Filling the buffer provides you with as much data as possible within the capture window. The buffer gets pre-filled with data samples prior to evaluating the trigger condition. As such, SignalTap requires that the buffer be filled at least once before any data can be retrieved through the JTAG connection 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 SignalTap II Logic Analyzer immediately evaluates all trigger conditions while writing samples into the acquisition memory. The logic analyzer evaluates each trigger condition before acquiring a full buffer's worth of samples. 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 until a full buffer's worth of data is captured 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 SignalTap II 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.

Figure 9-45 and Figure 9-46 show the difference between a non-segmented buffer in continuous mode and a non-segmented buffer using a storage qualifier. The logic analyzer for the waveforms below is configured with a sample depth of 64 bits, with a trigger position specified as **Post trigger position**.

Figure 9-45: SignalTap II Logic Analyzer Continuous Data Capture

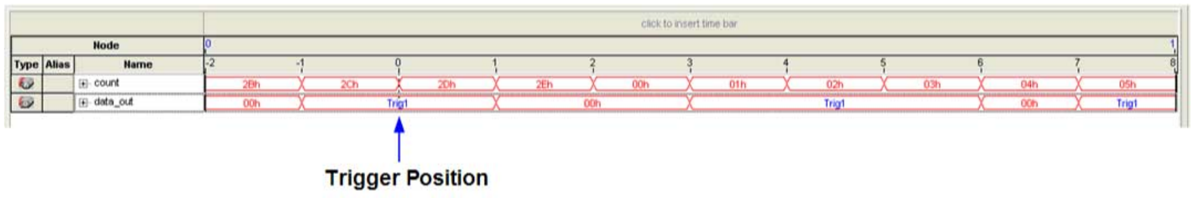


Note to Figure 9-45 :

1. Continuous capture mode with post-trigger position.
2. Capture of a recurring pattern using a non-segmented buffer in continuous mode. The SignalTap II Logic Analyzer is configured with a basic trigger condition (shown in the figure as "Trig1") with a sample depth of 64 bits.

Notice in Figure 9-45 that Trig1 occurs several times in the data buffer before the SignalTap II 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 9-46: SignalTap II Logic Analyzer Conditional Data Capture



Note to Figure 9-46 :

1. Conditional capture, storage always enabled, post-fill count.
2. SignalTap II Logic Analyzer capture of a recurring pattern using a non-segmented buffer in conditional mode. The logic analyzer is configured with a basic trigger condition (shown in the figure

as "Trig1"), with a sample depth of 64 bits. The “Trigger in” condition is specified as "Don't care", which means that every sample is captured.

Notice in Figure 9-46 that the logic analyzer triggers immediately. As in Figure 9-45, the logic analyzer completes the acquisition with eight samples, or 12% of 64, the sample capacity of the acquisition buffer.

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.

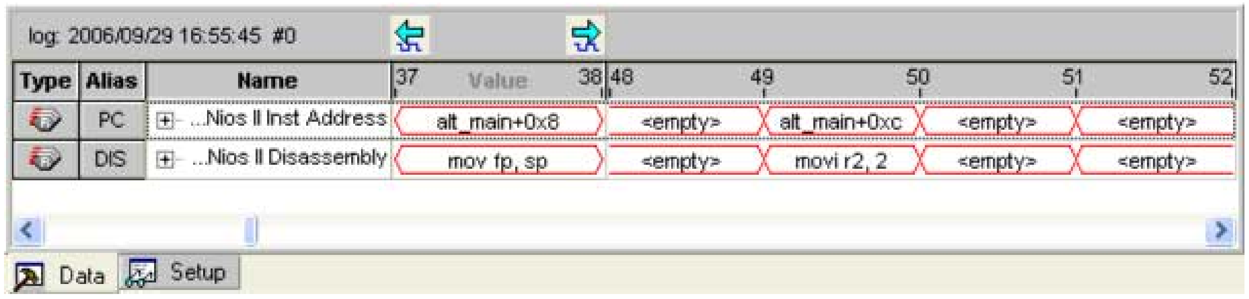
You use the labels you create in a table in different ways on the **Setup** and **Data** tabs. On the **Setup** tab, you can create basic triggers with meaningful names by right-clicking an entry in the **Trigger Conditions** column and selecting a label from the table you assigned to the signal group. 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.

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 on 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 9-47. Captured data samples around the trigger are referenced as offset addresses from the trigger function name.

Figure 9-47: Data Tab when the Nios II Plug-In is Used



log: 2006/09/29 16:55:45 #0

Type	Alias	Name	37 Value 38	48	49	50	51	52
	PC	...Nios II Inst Address	alt_main+0x8	<empty>	alt_main+0xc	<empty>	<empty>	
	DIS	...Nios II Disassembly	mov fp, sp	<empty>	movi r2, 2	<empty>	<empty>	

Locating a Node in the Design

When you find the source of an error in your design using the SignalTap II Logic Analyzer, you can use the node locate feature to locate that signal in many of the tools found in the 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 SignalTap II Logic Analyzer in one of the Quartus Prime software tools or your design files, right-click on 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

Saving Captured Data

The data log shows the history of captured data and the triggers used to capture the data. The SignalTap II Logic Analyzer acquires data, stores it in a log, and displays it as waveforms. When the logic analyzer is in auto-run mode and a trigger event occurs more than once, captured data for each time the trigger occurred is stored as a separate entry in the data log. This allows you to review the captured data for each trigger event. The default name for a log is based on the time when the data was acquired. Altera recommends that you rename the data log with a more meaningful name.

The logs are organized in a hierarchical manner; similar logs of captured data are grouped together in trigger sets. To open the **Data Log** pane, on the View menu, select **Data Log**. To turn on data logging, turn on **Enable data log** in the **Data Log** (**Figure 9-19**). To recall and activate a data log for a given trigger set, double-click the name of the data log in the list. The time stamping for the Data Log entries display the wall-clock time when SignalTap II triggered and the elapsed time from when acquisition started to when the device triggered.

Related Information

Managing Multiple SignalTap II Files and Configurations on page 9-21
You can use the Data Log feature for organizing different sets of trigger conditions and different sets of signal configurations.

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 SignalTap II Logic Analyzer, on the File menu, click **Export** and specify the **File Name**, **Export Format**, and **Clock Period**.

Creating a SignalTap II List File

Captured data can also be viewed in an **.stp** list file. An **.stp** list file is a text file that lists all the data captured by the logic analyzer for a trigger event. 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 a mnemonic table was created for the captured data, the numerical values in the list are replaced with a matching entry from the table. This is especially useful with the use of a plug-in that includes instruction code disassembly. You can immediately see the order in which the instruction code was executed during the same time period of the trigger event. To create an **.stp** list file in the Quartus Prime software, on the File menu, select **Create/Update** and click **Create SignalTap II List File**.

Other Features

The SignalTap II Logic Analyzer has other features that do not necessarily belong to a particular task in the task flow.

Using the SignalTap II MATLAB MEX Function to Capture Data

If you use MATLAB for DSP design, you can call the MATLAB MEX function `alt_signaltap_run`, built into the Quartus Prime software, to acquire data from the SignalTap II Logic Analyzer directly into a matrix in the MATLAB environment. 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 SignalTap II in the Quartus Prime software environment.

Note: The SignalTap II MATLAB MEX function is available in the Windows version and Linux version of the Quartus Prime software. It is compatible with MATLAB Release 14 Original Release Version 7 and later.

To set up the Quartus Prime software and the MATLAB environment to perform SignalTap II acquisitions, perform the following steps:

1. In the Quartus Prime software, create an **.stp** file.
2. In the node list in the **Data** tab of the SignalTap II 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 SignalTap II acquisition sample, while each row represents a signal or group of signals in the order they are organized in the **Data** tab.

   Note: Signal groups acquired from the SignalTap II Logic Analyzer and transferred into the MATLAB MEX function are limited to a width of 32 signals. If you want 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 32-signal limit.
3. Save the **.stp** and compile your design. Program your device and run the SignalTap II Logic Analyzer to ensure your trigger conditions and signal acquisition work correctly.
4. In the MATLAB environment, add the 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
   ```

Use the MATLAB MEX function to open the JTAG connection to the device and run the SignalTap II 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 Table 9-12.

Table 9-12: SignalTap II 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 SignalTap II **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 SignalTap II 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 SignalTap II data log if multiple configurations are present in the **.stp**. The default is the active signal set and trigger in the file.

You can enable or disable verbose mode to see the status of the logic analyzer while it is acquiring data. 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.

Using SignalTap II in a Lab Environment

You can install a stand-alone version of the SignalTap II 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 Quartus Prime installation, or if you do not have a license for a full installation of the Quartus Prime software. The standalone version of the SignalTap II Logic Analyzer is included with and requires the Quartus Prime stand-alone Programmer which is available from the Downloads page of the Altera website.

Remote Debugging Using the SignalTap II Logic Analyzer

Debugging Using a Local PC and an Altera SoC

You can use the System Console with SignalTap II Logic Analyzer to remote debug your Altera SoC. This method requires one local PC, an existing TCP/IP connection, a programming device at the remote location, and an Altera SoC.

Related Information

Remote Hardware Debugging over TCP/IP for Altera SoC application note

Debugging Using a Local PC and a Remote PC

You can use the SignalTap II 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 Quartus Prime software installed on the local PC
- Stand-alone SignalTap II Logic Analyzer or the full version of the 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

Equipment Setup

On the PC in the remote location, install the standalone version of the SignalTap II Logic Analyzer, included in the Quartus Prime standalone Programmer, or the full version of the Quartus Prime software. This remote computer must have Altera programming hardware connected, such as the EthernetBlaster or USB-Blaster.

On the local PC, install the full version of the Quartus Prime software. This local PC must be connected to the remote PC across a LAN with the TCP/IP protocol.

Using the SignalTap II 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 SignalTap II 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.

Altera 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 SignalTap II 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 SignalTap II Logic Analyzer. After the FPGA is configured with a SignalTap II Logic Analyzer instance in the design, when you open the SignalTap II Logic Analyzer in the Quartus Prime software, you then scan the chain and are ready to acquire data with the JTAG connection.

Backward Compatibility with Previous Versions of Quartus Prime Software

You can open an **.stp** created in a previous version in a current version of the Quartus Prime software. However, opening an **.stp** modifies it so that it cannot be opened in a previous version of the Quartus Prime software.

If you have a 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 SignalTap II Logic Analyzer. If you need to update your configuration, a prompt appears asking if you would like to update the **.stp** to match the current version of the Quartus Prime software.

SignalTap II Command-Line Options

To compile your design with the SignalTap II Logic Analyzer using the command prompt, use the `quartus_stp` command. Table 9-13 shows the options that help you use the `quartus_stp` executable.

Table 9-13: SignalTap II Command-Line Options

Option	Usage	Description
`stp_file`	`quartus_stp --stp_file <stp_filename>`	Assigns the specified **.stp** to the `USE_SIGNALTAP_FILE` in the **.qsf**.
`enable`	`quartus_stp --enable`	Creates assignments to the specified **.stp** in the **.qsf** and changes `ENABLE_SIGNALTAP` to `ON`. The SignalTap II Logic Analyzer is included in your design the next time the project is compiled. If no **.stp** is specified in the **.qsf**, the `--stp_file` option must be used. If the `--enable` option is omitted, the current value of `ENABLE_SIGNALTAP` in the **.qsf** is used.
`disable`	`quartus_stp --disable`	Removes the **.stp** reference from the **.qsf** and changes `ENABLE_SIGNALTAP` to `OFF`. The SignalTap II 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.

The first example illustrates how to compile a design with the SignalTap II 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 Quartus Prime software to compile the **stp1.stp** file with your design. The `--enable` option must be applied for the SignalTap II Logic Analyzer to compile properly into your design.

The example below shows how to create a new **.stp** after building the SignalTap II Logic Analyzer instance with the IP Catalog.

```
quartus_stp filtref --create_signaltap_hdl_file --stp_file stp1.stp
```

Related Information

Command-Line Scripting documentation
Information about the other command line executables and options

SignalTap II Tcl Commands

The **quartus_stp** executable supports a Tcl interface that allows you to capture data without running the Quartus Prime GUI. You cannot execute SignalTap II Tcl commands from within the Tcl console in the Quartus Prime software. They must be executed from the command line with the **quartus_stp** executable. To execute a Tcl file that has SignalTap II Logic Analyzer Tcl commands, use the following command:

```
quartus_stp -t <Tcl file>
```

The example is an excerpt from a script you can use to continuously capture data. Once the trigger condition is met, the data is captured and stored in the data log.

```
#opens signaltap session
open_session -name stp1.stp
#start acquisition of instance auto_signaltap_0 and
#auto_signaltap_1 at the same time
#calling run_multiple_end will start all instances
#run after run_multiple_start call
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 signaltap session
close_session
```

When the script is completed, open the **.stp** that you used to capture data to examine the contents of the Data Log.

Related Information

::quartus::stp online help
Information about Tcl commands that you can use with the SignalTap II Logic Analyzer Tcl package

Design Example: Using SignalTap II 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. In this example, the Nios processor executes a simple C program from on-chip memory and waits for you to press a button. After you press a button, the processor initiates a DMA transfer, which you analyze using the SignalTap II Logic Analyzer.

Related Information

AN 446: Debugging Nios II Systems with the SignalTap II Embedded Logic Analyzer application note

Custom Triggering Flow Application Examples

The custom triggering flow in the SignalTap II 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 SignalTap II 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 Information

On-chip Debugging Design Examples website

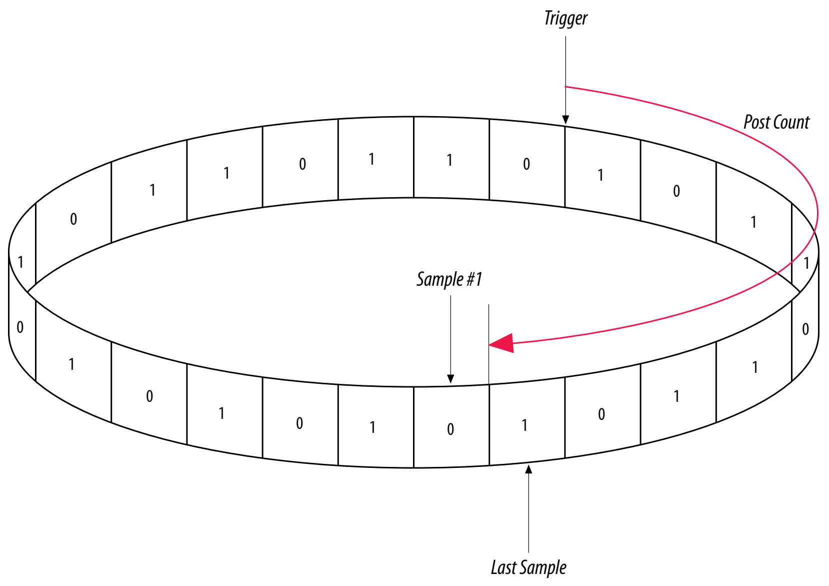
Design Example 1: Specifying a Custom Trigger Position

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 last acquisition before stopping the buffer is displayed on the **Data** tab 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 9-48 illustrates the triggering position.

Figure 9-48: Specifying a Custom Trigger Position



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 SignalTap II **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;
```

SignalTap II 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 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 Information

- Tcl Scripting documentation
- Quartus Prime Tcl Scripting online help

Document Revision History

Table 9-14: Document Revision History

Date	Version	Changes Made
2015.11.02	15.1.0	• Changed instances of *Quartus II* to *Quartus Prime*. • Updated content to reflect SignalTap support in Quartus Prime Pro Edition
2015.05.04	15.0.0	Added content for Floating Point Display Format in table: SignalTap II 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 SignalTap II 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.

Date	Version	Changes Made
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 SignalTap II 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 SignalTap II software.
December 2010	10.0.1	Changed to new document template.
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 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 SignalTap II Logic Analyzer" on page 13–45 • Added new Figure 13–33 • Made minor editorial updates
November 2008	8.1.0	Updated for the 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

Date	Version	Changes Made
May 2008	8.0.0	Updated for the 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 Information

Quartus Handbook Archive

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.

Debugging Single Event Upset Using the Fault Injection Debugger 10

2015.11.02

QPP5V3 Subscribe Send Feedback

You can detect and debug single event upset (SEU) using the Fault Injection Debugger in the Quartus® Prime software. Use the debugger with the Altera Fault Injection IP core to inject errors into the configuration RAM (CRAM) of an FPGA device.

The injected error simulate the soft errors that can occur during normal operation due to (SEUs). Because 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 Stratix V family devices. For assistance with support for Arria V or Cyclone V family devices, file a service request using **mySupport**.

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 Information

Altera Website: Single Event Upsets

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.

Arria V, Cyclone V, Stratix V and newer devices have the following CRAM capabilities:

- Error Detection Cyclical Redundancy 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 Altera devices, refer to the *SEU Mitigation* chapter in the respective device handbook.

Related Information

Altera Website: Single Event Upsets

Hardware and Software Requirements

The following hardware and software is required to use the Fault Injection Debugger:

- Quartus Prime software version 14.0 or later.
- FEATURE line in your Altera license that enables the Fault Injection IP core. For more information, contact your local Altera sales representative.
- Download cable (USB-Blaster, USB-Blaster II, EthernetBlaster, or EthernetBlaster II cable).
- Altera development kit or user designed board with a JTAG connection to the device under test.
- (Optional) FEATURE line in your Altera license that enables the Advanced SEU Detection IP core.

Related Information

Altera Website: Contact Altera

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 Quartus Prime software or from the command line to simulate soft errors.

The Fault Injection Debugger communicates with the Fault Injection IP core via the JTAG interface. You perform debugging using the Fault Injection Debugger in the Quartus Prime software or using the command-line interface.

- 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.

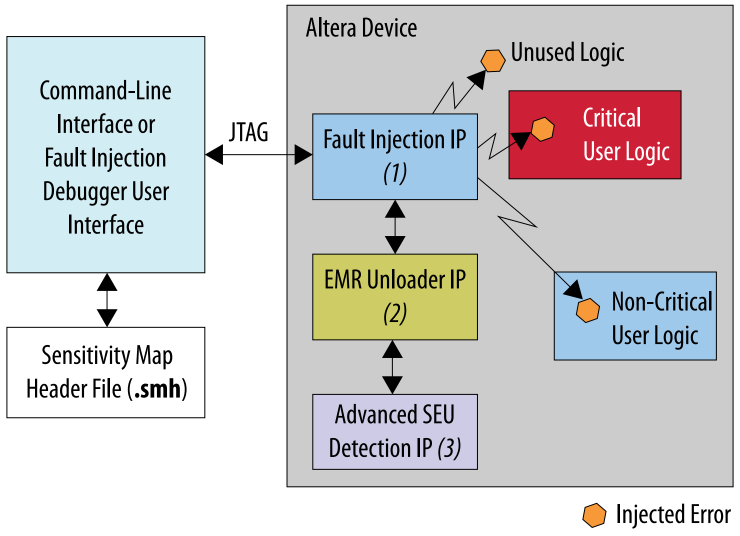
The Fault Injection IP accepts commands from the JTAG interface and reports status back through the JTAG interface.

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 Altera 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 10-1: Fault Injection Debugger Overview Block Diagram


Command-Line Interface or Fault Injection Debugger User Interface
JTAG
Sensitivity Map Header File (.smh)
Altera Device
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 Information

- Download Center
- AN 539: Test Methodology or Error Detection and Recovery using CRC in Altera FPGA Devices
- Understanding Single Event Functional Interrupts in FPGA Designs White Paper
- Altera Fault Injection IP Core User Guide
- Altera Error Message Unloader IP Core User Guide
- Altera Advanced SEU Detection (ALTERA_ADV_SEU_DETECTION) IP Core User Guide

Instantiating the 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 Qsys 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 Qsys and the IP Catalog. Optionally, you can instantiate them directly into your RTL design, using Verilog HDL, SystemVerilog, or VHDL.

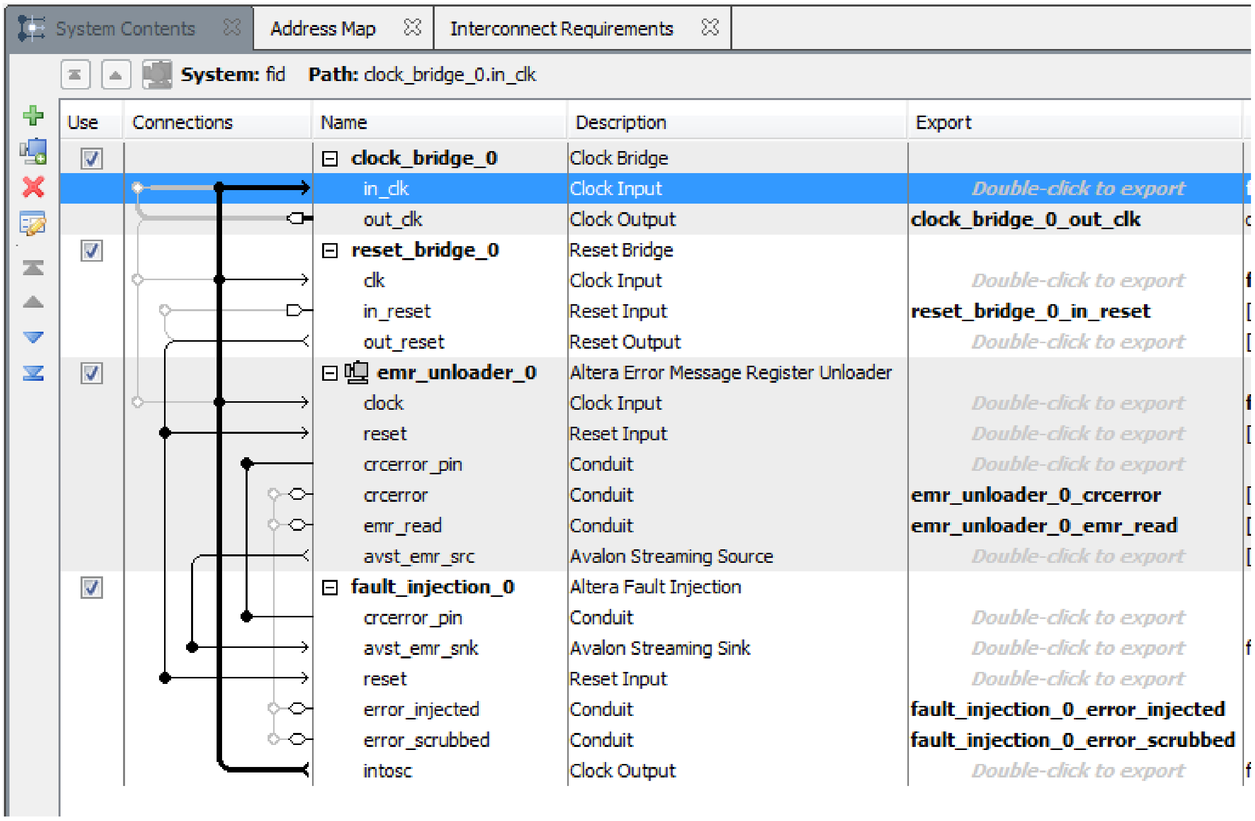
Related Information

- Altera Fault Injection IP Core User Guide

Using 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 10-2: Example Qsys System Including the Fault Injection IP Core and EMR Unloader IP Core

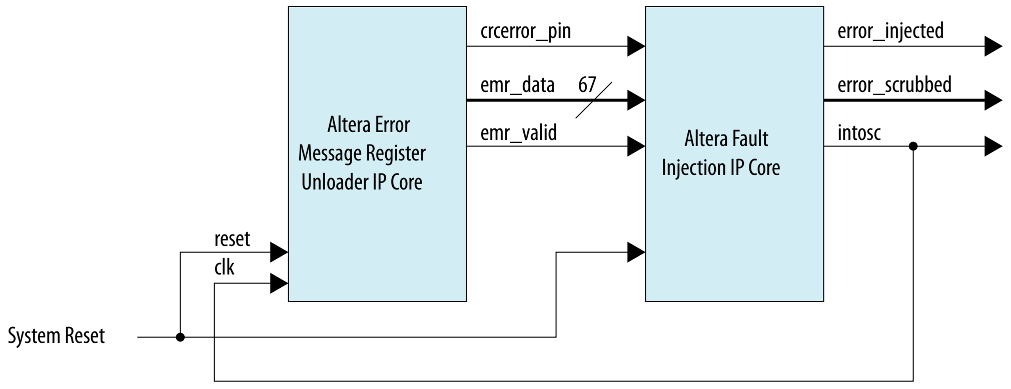


System Contents | Address Map | Interconnect Requirements

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 10-3: Example Altera Fault Injection IP Core and EMR Unloader IP Core Block Diagram



Related Information

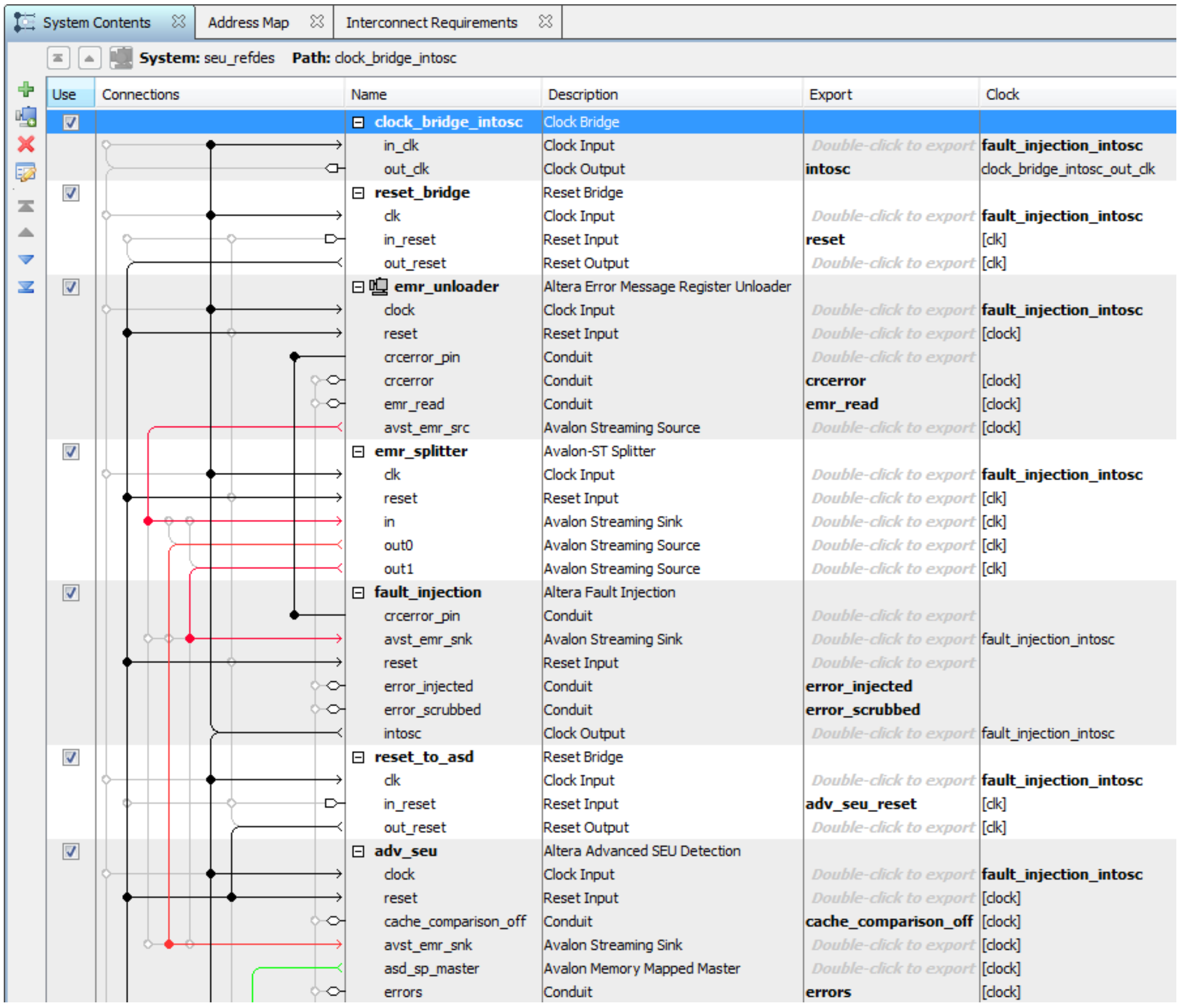
- [Altera Error Message Unloader IP Core User Guide]()

Using 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 Qsys system in which an Avalon-ST splitter distributes the EMR contents to the ASD and Fault Injection IP cores.

Figure 10-4: Using the ASD and Fault Injection IP in the Same Qsys System



System Contents | Address Map | Interconnect Requirements

System: seu_refdes **Path:** clock_bridge_intosc

Use	Connections	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 Information

- **Altera Advanced SEU Detection (ALTERA_ADV_SEU_DETECTION) IP Core User Guide**

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 Quartus Prime Assembler generates the SMH file. The Fault Injection Debugger limits error injections to specific device regions you define in the SMH file.

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.

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.

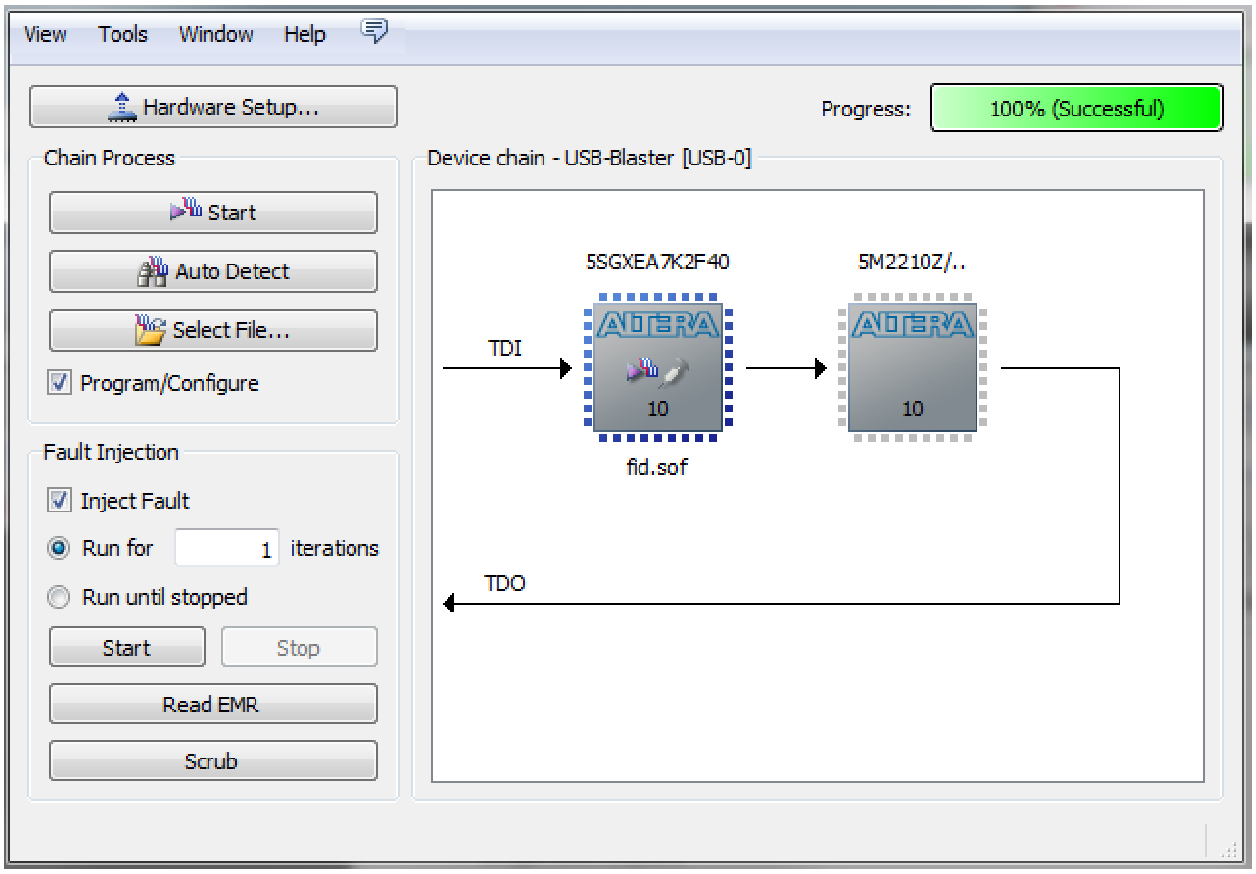
Using the Fault Injection Debugger

To use the Fault Injection Debugger, you connect from the tool to the 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 Quartus Prime software.

Note: Configuring or programming the device is similar to the procedure used for the Programmer or SignalTap II Logic Analyzer.

Figure 10-5: 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 Information

Targeted Fault Injection Feature on page 10-13

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**:

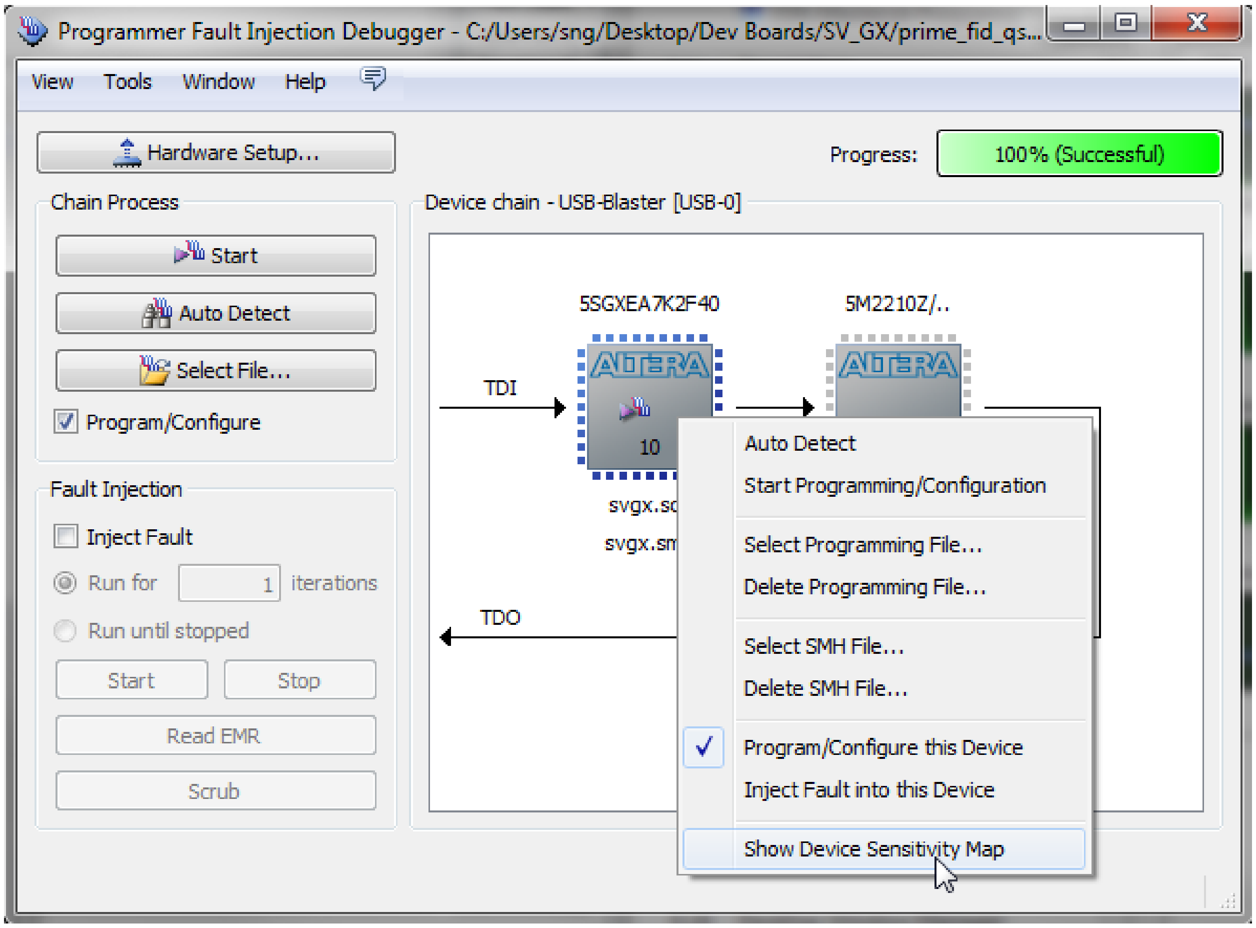
1. Select the FPGA you wish to configure in the **Device chain** box.
2. Click **Select File**.
3. Navigate to the **.sof** and click **OK**. The Fault Injection Debugger reads the **.sof**.
4. (Optional) Select the SMH file.

   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 10-6: Context Menu for Selecting the SMH File



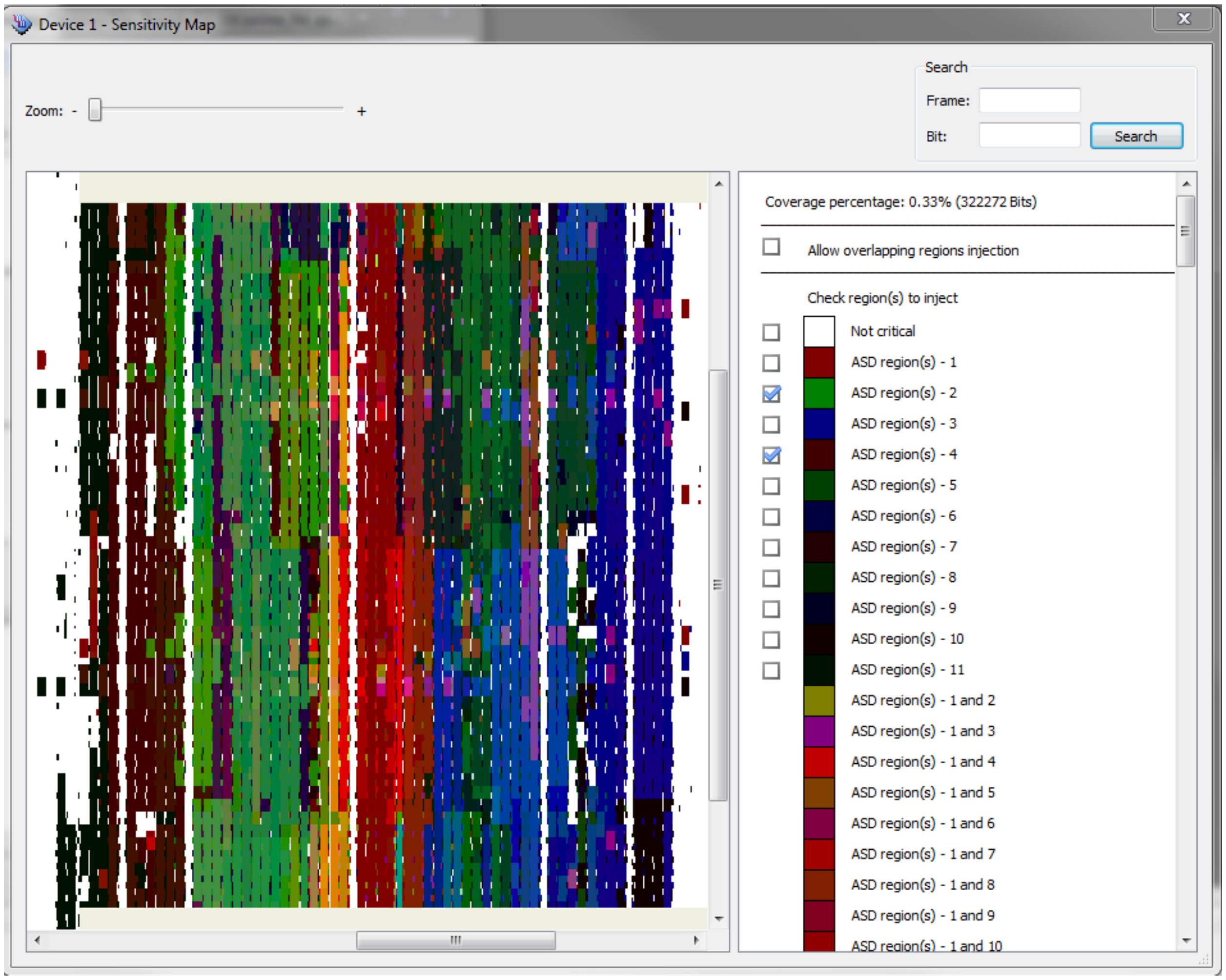
Constraining Regions for Fault Injection

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 on the FPGA in the **Device chain** box and click Show Device Sensitivity Map.
2. Select the ASD region(s) for fault injection.

Figure 10-7: 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

Specifying Error Types

You can specify various types of errors for injection.

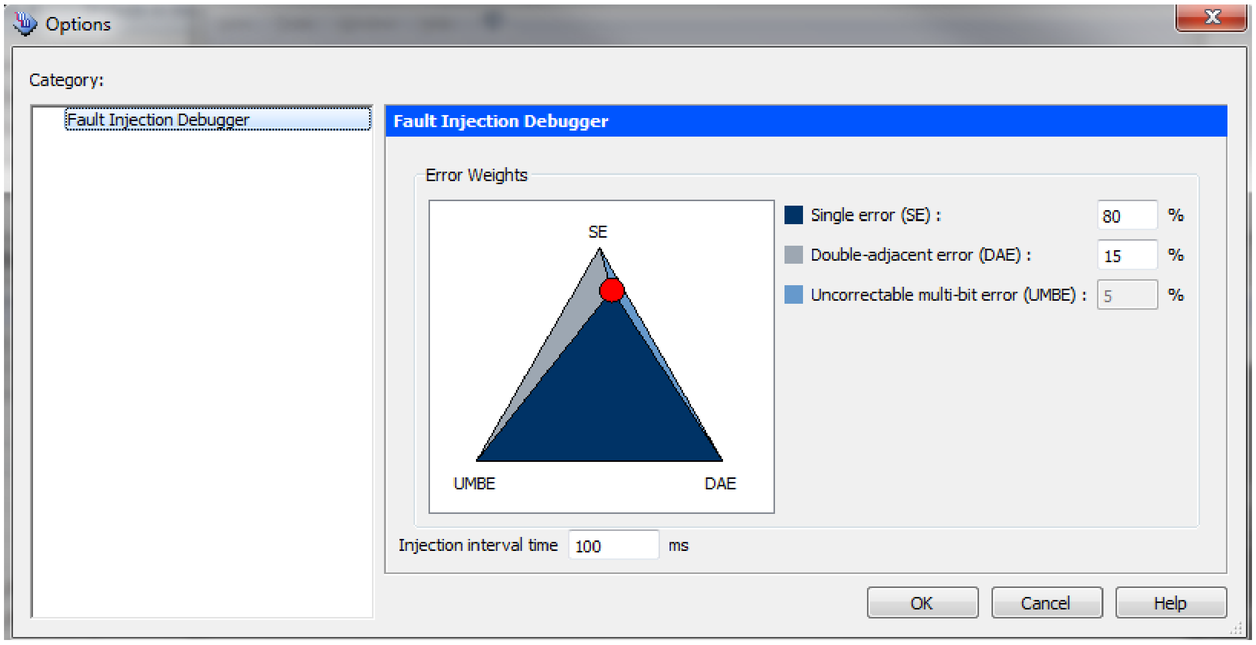
- Single errors (SE)
- Double-adjacent errors (DAE)
- Uncorrectable multi-bit errors (EMBE)

Altera devices can self-correct single and double-adjacent errors if the scrubbing feature is enabled. Altera 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 10-8: 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 Information

Mitigating Single Event Upsets

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

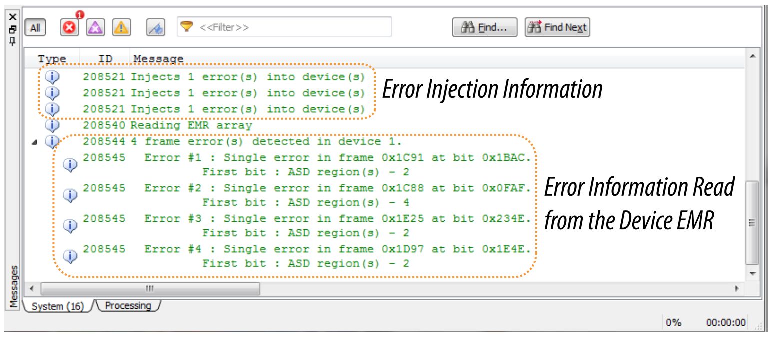
To inject these faults:

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 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 10-9: 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
```

Recording Errors

You can record the location of any injected fault by noting the parameters reported in the 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.

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.

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 10-1: 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: 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: 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: 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: 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: 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
```

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 Information

AN 539: Test Methodology or Error Detection and Recovery using CRC in Altera FPGA Devices

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.

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

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 Information

AN 539: Test Methodology or Error Detection and Recovery using CRC in Altera FPGA Devices

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

Document Revision History

Table 10-2: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *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 Require-ments” section.
December 2012	2012.12.01	Preliminary release.

In-System Debugging Using External Logic Analyzers 11

2015.11.02

QPP5V3 Subscribe Send Feedback

About the Quartus Prime Logic Analyzer Interface

The Quartus Prime Logic Analyzer Interface (LAI) allows you to use an external logic analyzer and a minimal number of Altera-supported device I/O pins to examine the behavior of internal signals while your design is running at full speed on your Altera®- 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 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 Altera-supported device. Instead of having a one-to-one relationship between internal signals and output pins, the 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 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 Information

Device Support website

Choosing a Logic Analyzer

The 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 SignalTap® II Logic Analyzer
- An external logic analyzer, which connects to internal signals in your Altera-supported device by using the Quartus Prime LAI

Table 11-1: Comparing the SignalTap II 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 SignalTap II Logic Analyzer, the maximum sample depth is set to 128 Kb, which is a device constraint. However, with an external logic analyzer, there are no device constraints, providing you a wider sample depth.	LAI
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 SignalTap II Logic Analyzer with your design. An external logic analyzer adds minimal logic, which removes resource limits on place-and-route.	LAI
Triggering Capability	The SignalTap II Logic Analyzer offers triggering capabilities that are comparable to external logic analyzers.	LAI or SignalTap II
Use of Output Pins	Using the SignalTap II Logic Analyzer, no additional output pins are required. Using an external logic analyzer requires the use of additional output pins.	SignalTap II
Acquisition Speed	With the SignalTap II 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.	SignalTap II

Related Information

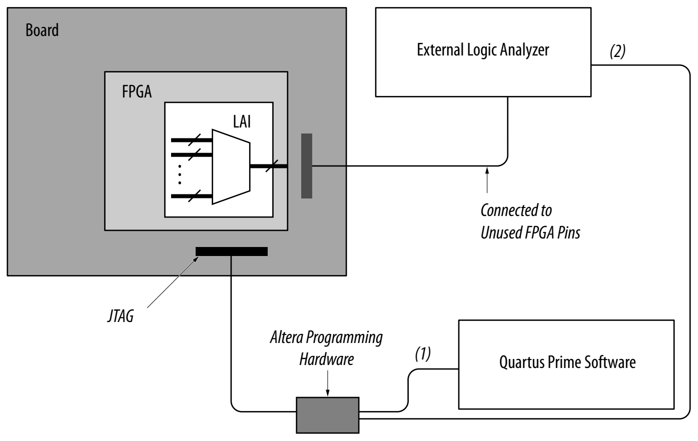
- **System Debugging Tools Overview** on page 5-1
  Overview and comparison of all tools available in the Quartus Prime software on-chip debugging tool suite

Required Components

You must have the following components to perform analysis using the LAI:

- The Quartus Prime software starting with version 5.1 and later
- The device under test
- An external logic analyzer
- An Altera communications cable
- A cable to connect the Altera-supported device to the external logic analyzer

Figure 11-1: LAI and Hardware Setup

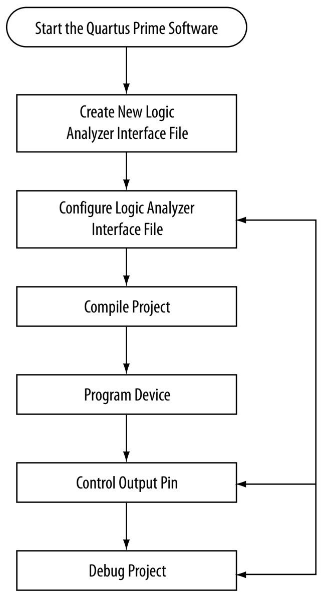


Notes to figure:

1. Configuration and control of the LAI using a computer loaded with the 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.

Flow for Using the LAI

Figure 11-2: 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 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.

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.

Configuring the File Core Parameters

After you create the **.lai** file, you must 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 11-2: 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 Altera-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.

Parameter	Description
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 Altera-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 Altera-supported device, it provides you with a functional view of your Altera-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, Altera 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.

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 the Pin Planner opens.

Related Information

Managing Device I/O Pins documentation
Information about how to use the Pin Planner

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**.

Using the Node Finder

Before making bank assignments, on the View menu, point to **Utility Windows** and click **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 **SignalTap II: pre-synthesis** for non-incrementally routed instances and **SignalTap II: 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** of your **.lai** file begins to reflect your assignments. 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.

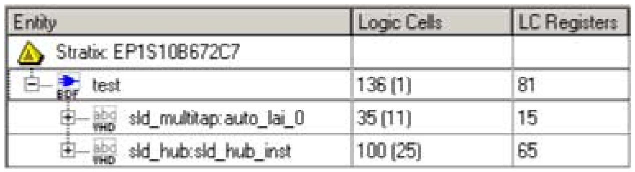
Compiling Your Quartus Prime Project

When you save your **.lai** file, a dialog box prompts you to enable the LAI instance for the active project. Alternatively, you can specify the **.lai** file your project uses in the **Global Project Settings** dialog box.

After you specify the name of your **.lai** file, you must compile your project. To compile your project, on the Processing menu, click **Start Compilation**.

To ensure that the LAI is properly compiled with your project, expand the entity hierarchy in the Project Navigator. (To display the Project Navigator, on the View menu, point to **Utility Windows** and click **Project Navigator**.) If the LAI is compiled with your design, the `sld_hub` and `sld_multitap` entities are shown in the Project Navigator.

Figure 11-3: 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

Programming Your Altera-Supported Device Using the LAI

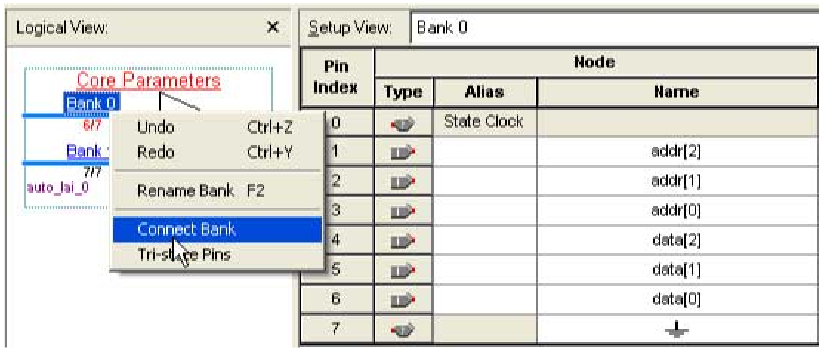
After compilation completes, you must configure your Altera-supported device before using the LAI.

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-Altera, JTAG-compliant devices. To use the LAI in more than one Altera-supported device, create an **.lai** file and configure an **.lai** file for each Altera-supported device that you want to analyze.

Controlling the Active Bank During Runtime

When you have programmed your Altera-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 11-4: Configuring Banks



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	◆		⏚

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.

Document Revision History

Table 11-3: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *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 Quartus Prime Help • Editorial updates • Removed Referenced Documents section
November 2009	9.1.0	• Removed references to APEX devices • Editorial updates

Date	Version	Changes
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 Information

Quartus Handbook Archive

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.

In-System Modification of Memory and Constants 12

2015.11.02

QPP5V3 Subscribe Send Feedback

About the In-System Memory Content Editor

The Quartus Prime In-System Memory Content Editor allows you to view and update memories and constants with the JTAG port connection.

The In-System Memory Content Editor allows access to dense and complex FPGA designs. When you program devices, you have read and write access to the memories and constants 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 SignalTap II Logic Analyzer, you can more easily view and debug your design in the hardware lab.

The ability to read data from memories and constants allows you to quickly 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. You can also intentionally write incorrect parity bit values into your RAM to check the error handling functionality of your design.

Related Information

System Debugging Tools Overview on page 5-1
Overview and comparison of all tools available in the Quartus Prime software on-chip debugging tool suite

Design Debugging Using the SignalTap II Logic Analyzer documentation on page 9-1

Library of Parameterized Modules online help
List of the types of memories and constants currently supported by the Quartus Prime software

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.

Creating In-System Modifiable Memories and Constants

When you specify that a memory or constant is run-time modifiable, the 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>";
```

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 Altera® 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 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.

Instance Manager

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.

You can read and write to in-system memory with the **Instance Manager** pane.

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 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 Information

Instance Manager Pane online help

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.

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.

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 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 Information

- Tcl Scripting documentation
- Command-Line Scripting documentation
- API Functions for Tcl online help
  Descriptions of the command options and scripting examples

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.

Example: Using the In-System Memory Content Editor with the SignalTap II Logic Analyzer

The following scenario describes how you can use the In-System Updating of Memory and Constants feature with the SignalTap II Logic Analyzer to efficiently debug your design. You can use the In-System Memory Content Editor and the SignalTap II 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 SignalTap II Logic Analyzer.
2. Using the SignalTap II 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 SignalTap II 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.

Document Revision History

Table 12-1: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *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 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 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 Information

Quartus Handbook Archive

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.

Design Debugging Using In-System Sources and Probes 13

2015.11.02

QPP5V3 Subscribe Send Feedback

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. The SignalTap® II Logic Analyzer and SignalProbe allow you to read or "tap" internal logic signals during run time as a way to debug your logic design.

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:

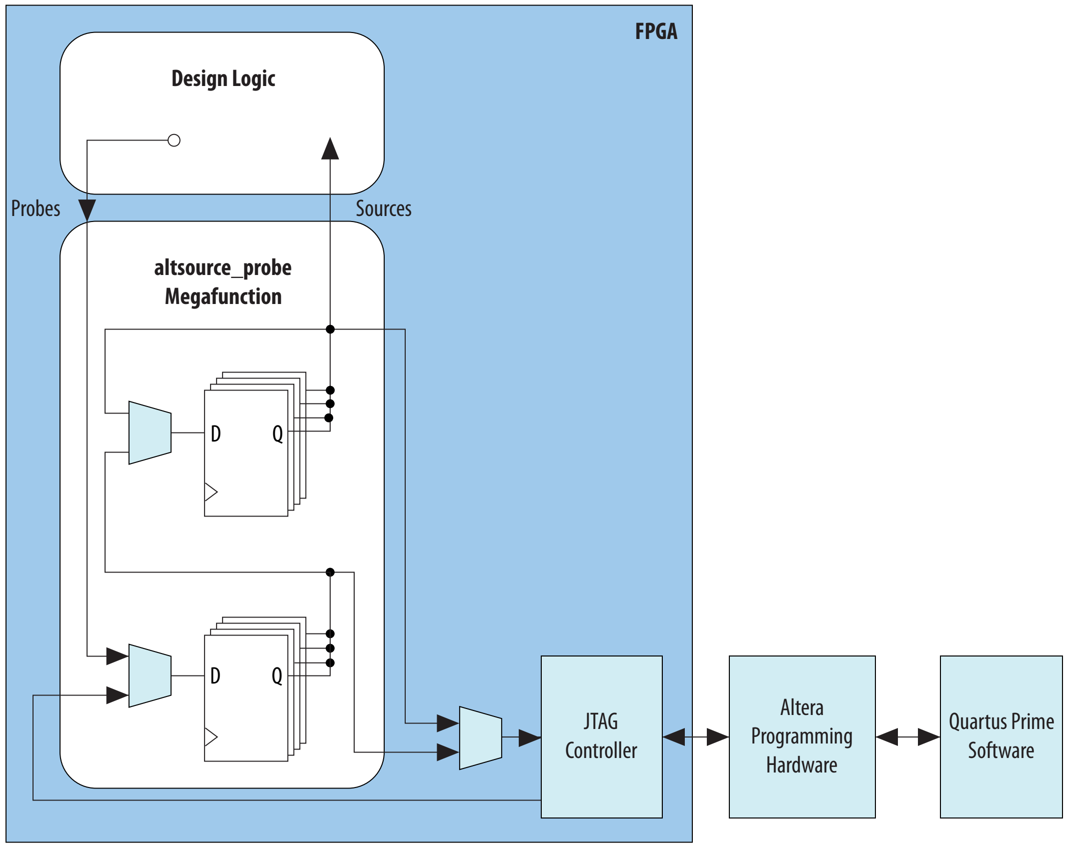
- Force the occurrence of trigger conditions set up in the SignalTap II Logic Analyzer
- Create simple test vectors to exercise your design without using external test equipment
- Dynamically control run time control signals with the JTAG chain

The In-System Sources and Probes Editor in the 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 SignalTap II Logic Analyzer or SignalProbe, 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 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 13-1: 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 SignalTap II 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 Information

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

Hardware and Software Requirements

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

- Quartus Prime software

or

- Quartus Prime Lite Edition (with the TalkBack feature turned on)
- Download Cable (USB-Blaster™ download cable or ByteBlaster™ cable)
- Altera® 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:

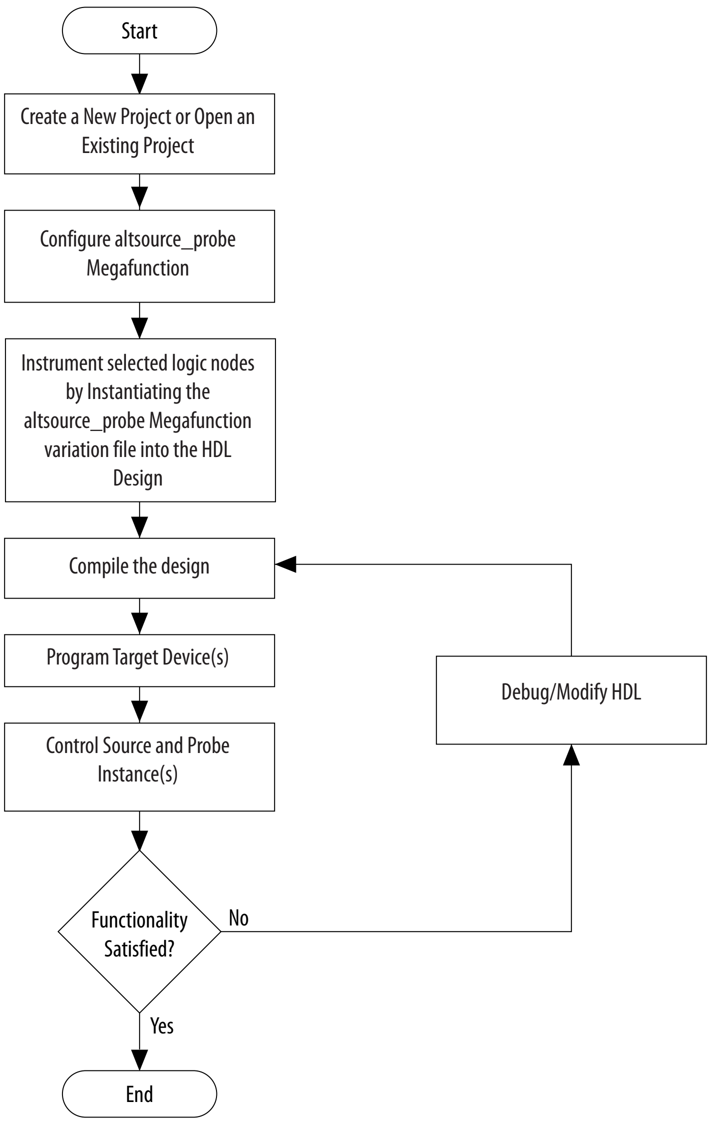
- Arria® series
- Stratix® series
- Cyclone® series
- MAX® series

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 13-2: 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

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.

In-System Sources and Probes IP Core Parameters

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

Table 13-1: 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.

When you compile your design that includes the In-System Sources and Probes IP 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.

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.

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.**

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 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.

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.

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.
- **Write Source Data**—Writes data to all source nodes of the selected instance.
- **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.

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.

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.

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**.

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.

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.

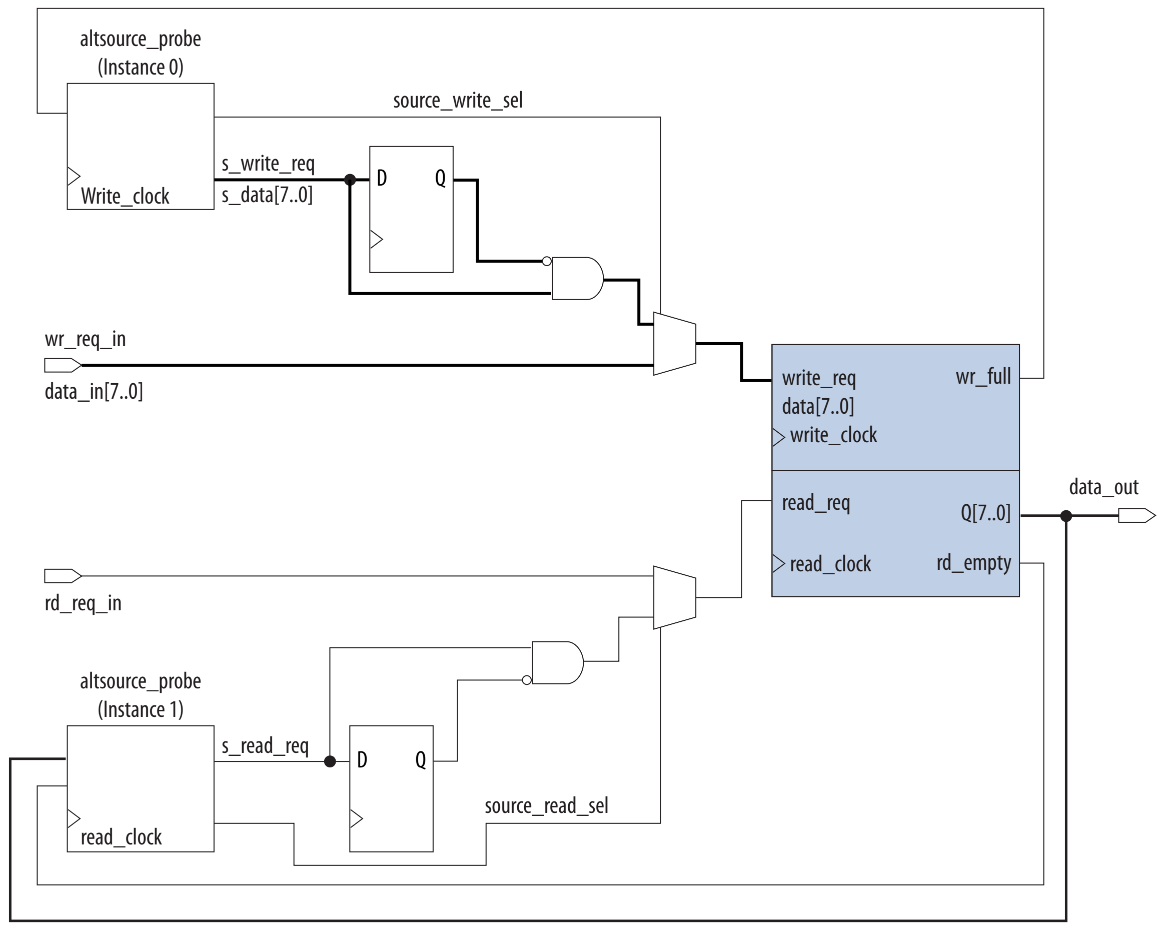
Table 13-2: In-System Sources and Probes Tcl Commands

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_interactive_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 SignalTap II Logic Analyzer.

Figure 13-3: 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 Information

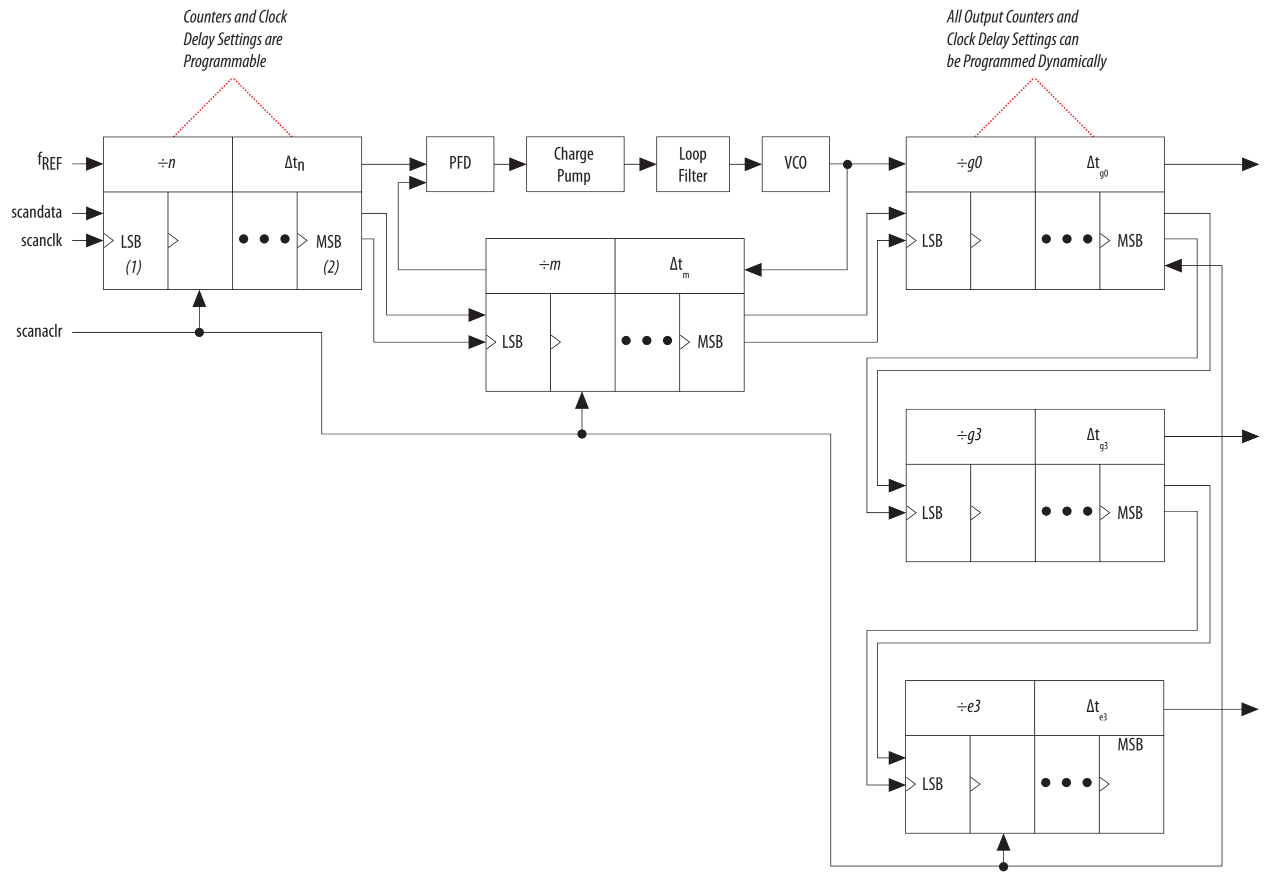
- Tcl Scripting
- Quartus Prime Settings File Manual
- Command Line Scripting

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.

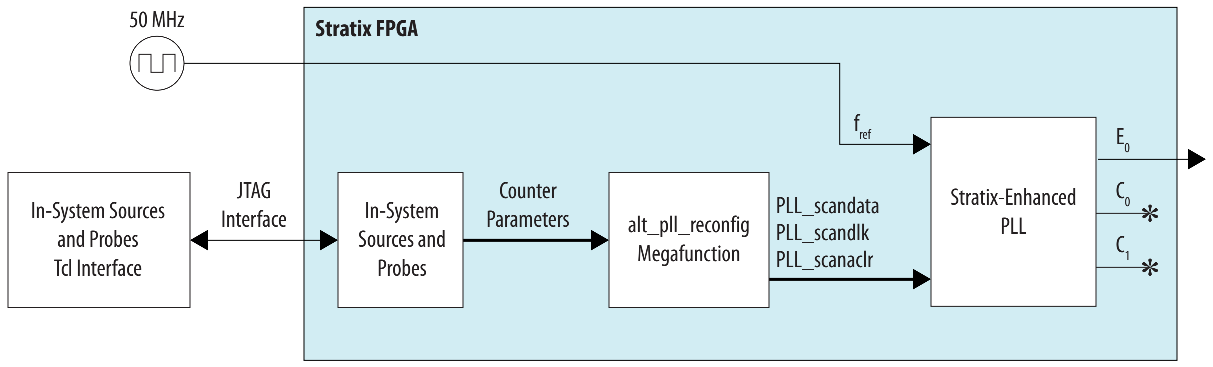
Stratix PLLs allow you to dynamically update PLL coefficients during run time. Each enhanced PLL within the Stratix device contains a register chain that allows you to modify the pre-scale counters (m and n values), output divide counters, and delay counters. In addition, the ALTPLL_RECONFIG IP core 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 13-4: 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
LSB
MSB
÷g3
Δtg3
LSB
MSB
÷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.

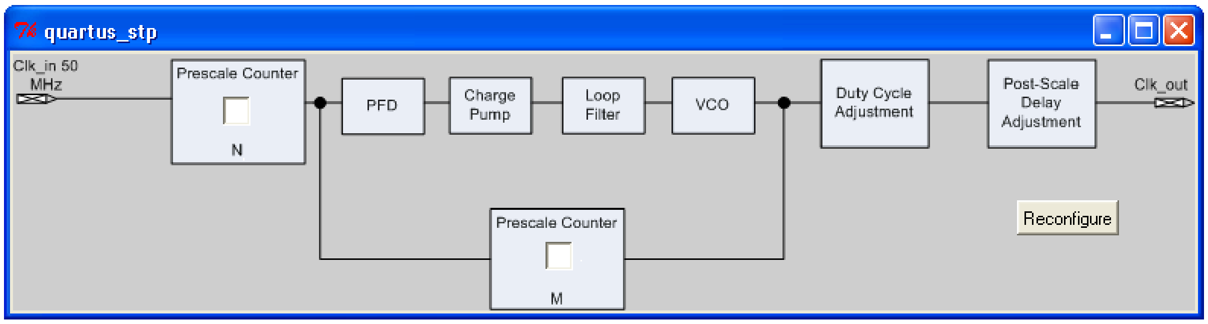
Figure 13-5: Block Diagram of Dynamic PLL Reconfiguration Design Example



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 Quartus Prime project for this design example.

Figure 13-6: Interactive PLL Reconfiguration GUI Created with Tk and In-System Sources and Probes Tcl Package



Related Information

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

Document Revision History

Table 13-3: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Quartus Prime*.
June 2014	14.0.0	Updated formatting.
June 2012	12.0.0	Removed survey link.

Date	Versio n	Changes
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 Information

Quartus Prime Handbook Archive
For previous versions of the Quartus Prime Handbook

Programming Altera Devices 14

2015.11.02

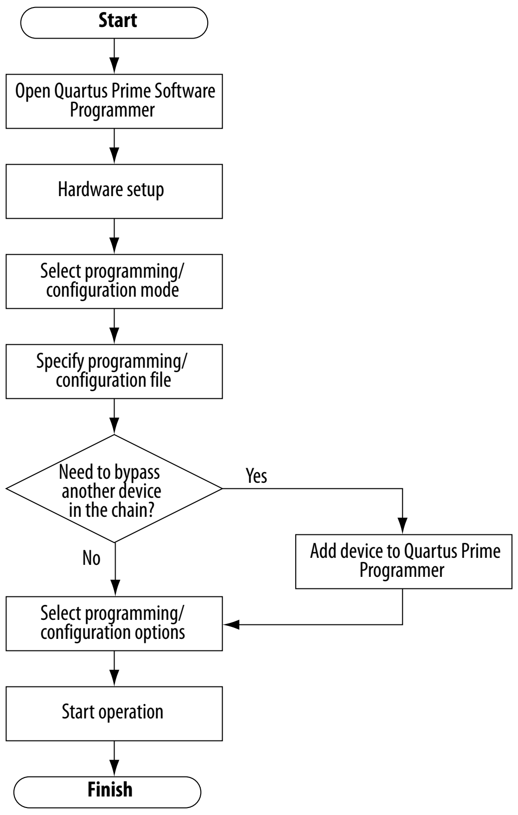
The Quartus Prime Programmer allows you to program and configure Altera® CPLD, FPGA, and configuration devices. After compiling your design, use the Quartus Prime Programmer to program or configure your device, to test the functionality of the design on a circuit board.

Related Information

- Programming Devices

Programming Flow

Figure 14-1: Programming Flow


Start
Open Quartus Prime Software Programmer
Hardware setup
Select programming/ configuration mode
Specify programming/ configuration file
Need to bypass another device in the chain?
Yes
No
Add device to Quartus Prime Programmer
Select programming/ configuration options
Start operation
Finish

The following steps describe the programming flow:

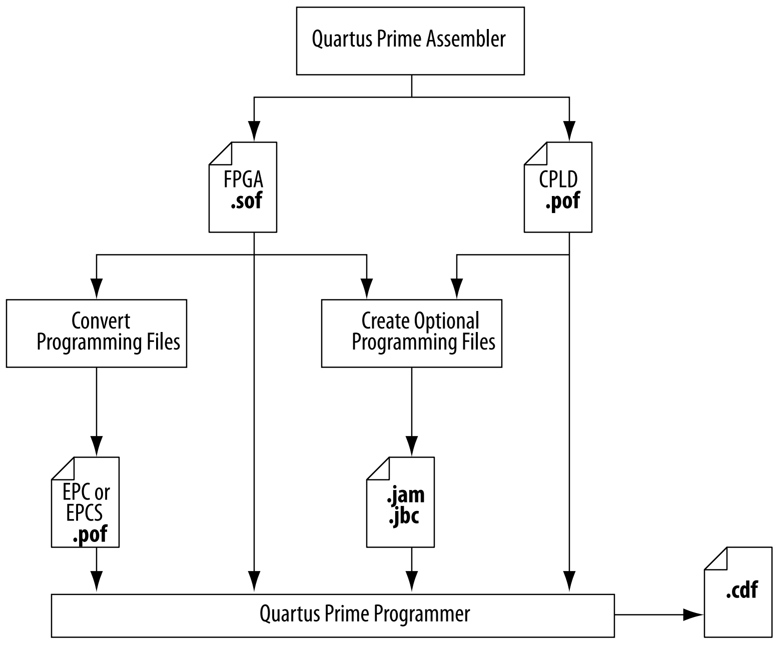
1. Compile your design, such that the Quartus Prime Assembler generates the programming or configuration file.
2. Convert the programming or configuration file to target your configuration device and, optionally, create secondary programming files.

Table 14-1: 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. Program and configure the FPGA, CPLD, or configuration device using the programming or configuration file with the Quartus Prime Programmer.

Figure 14-2: Programming File Generation Flow



Stand-Alone Quartus Prime Programmer

Altera offers the free stand-alone Programmer, which has the same full functionality as the Quartus Prime Programmer in the 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 14-2: 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 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 Altera devices.

Related Information

- **Download Center**
  You can download the stand-alone Quartus Prime Programmer from this page.

Optional Programming or Configuration Files

The Quartus Prime software can generate optional programming or configuration files in various formats that you can use with programming tools other than the Quartus Prime Programmer. When you compile a design in the 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 Information

- **AN 425: Using Command-Line Jam STAPL Solution for Device Programming**

  Describes how to use the **.jam** and **.jbc** programming files with the Jam STAPL Player, Jam STAPL Byte-Code Player, and the `quartus_jli` command-line executable.

Secondary Programming Files

The Quartus Prime software generates programming files in various formats for use with different programming tools.

Table 14-3: File Types Generated by the Quartus Prime Software and Supported by the Quartus Prime Programmer

File Type	Generated by the Quartus Prime Software	Supported by the Quartus Prime Programmer
.sof	Yes	Yes

File Type	Generated by the Quartus Prime Software	Supported by the Quartus Prime Programmer
.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 [8]
Tabular Text File (**.ttf**)	Yes	—
Raw Programming Data File (**.rpd**)	Yes	—

Quartus Prime Programmer GUI

The Quartus Prime Programmer GUI is a window that allows you to perform the following tasks:

- Adding your programming and configuration files.
- Specifying programming options and hardware.
- Starting the programming or configuration of the device.

To open the Programmer window, on the Tools menu, click **Programmer**. As you proceed through the programming flow, the Quartus Prime Message window reports the status of each operation.

Related Information

- **Programmer Page (Options Dialog Box)**
  Describes the options in the **Tools** menu.

Editing the Device Details of an Unknown Device

If the Quartus Prime Programmer automatically detects devices with shared JTAG IDs, the Programmer prompts you to specify the correct device in the JTAG chain.

If the Programmer does not prompt you to specify the correct device in the JTAG chain, then you must add a user defined device in the Quartus Prime software for each unknown device in the JTAG chain and specify the instruction register length for each device.

[8] Raw Binary File (**.rbf**) is supported by the Quartus Prime Programmer in Passive Serial (PS) configuration mode.

To edit the device details of an unknown device, follow these steps:

1. Double-click on the unknown device listed under the device column.
2. Click **Edit**.
3. Change the device **Name**.
4. Enter the **Instruction register Length**.
5. Click **OK**.
6. Save the **.cdf**.

Setting Up Your Hardware

The Quartus Prime Programmer provides the flexibility to choose a download cable or programming hardware. Before you can program or configure your device, you must have the correct hardware setup.

Related Information

- **Setting up Programming Hardware in Quartus Prime Software**
  Describes the programming hardware driver installation.

Setting the JTAG Hardware

The JTAG server allows the 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.

Running JTAG Daemon with Linux

The JTAGD daemon is the Linux version of a JTAG server. The JTAGD daemon allows a board which is connected to a Linux host to be programmed or debugged over the network from a remote machine. The JTAGD daemon also allows multiple programs to use JTAG resources at the same time.

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.

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 from the Tools menu on the main menu of the Quartus Prime software.

Programming and Configuration Modes

The following table lists the programming and configuration modes supported by Altera devices.

Table 14-4: Programming and Configuration Modes

Configuration Mode Supported by the 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 Information

- **Configuration via Protocol (CvP) Implementation in Altera FPGAs User Guide**
  Describes the CvP configuration mode.
- **Programming Adapters**
  Contains a list of programming adapters available for Altera devices.

Design Security Keys

The Quartus Prime Programmer supports the generation of encryption key programming files and encrypted configuration files for Altera FPGAs that support the design security feature. You can also use the Quartus Prime Programmer to program the encryption key into the FPGA.

Related Information

- **AN 556: Using the Design Security Features in Altera FPGAs**

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. 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**. To save time in subsequent conversions, you can 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.

To access the **Convert Programming Files** dialog box, on the main menu of the Quartus Prime software, click **File** > **Convert Programming Files**.

Example 14-1: 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 Information

- Convert Programming Files Dialog Box

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 Altera device. You can direct the 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 14-5: 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). 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.

Option Setting	Description
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 [9]	Yes [10]	—
Decompression feature is enabled.	—	Yes	Yes	Yes [9]	Yes [10]	—
Encryption feature is enabled.	—	Yes	Yes	Yes [9]	Yes [10]	—
`CONF_DONE` stays low after a configuration cycle.	—	Yes	Yes [11]	Yes [9]	Yes [10]	—

[9] Use only for multi-device chain
[10] Use only for single-device chain
[11] Start with positive offset to the PLC settings

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` goes high momentarily after a configuration cycle.	—	Yes	Yes [12]	—	—	—
FPGA does not enter user mode even though `CONF_DONE` goes high.	—	—	—	Yes [9]	Yes [10]	—
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 Quartus Prime Convert Programming File Utility when the decompression feature is enabled.	—	—	—	—	—	Yes

[12] Start with negative offset to the PLC settings

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 Quartus Prime software supports parallel and serial flash loaders.

Scripting Support

In addition to the Quartus Prime Programmer GUI, you can use the Quartus Prime command-line executable `quartus_pgm.exe` to access programmer functionality from the command line and from scripts. The programmer accepts **.pof**, **.sof**, and **.jic** programming or configuration files and **.cdf**.

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 ByteBlaster II 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.

Note: For linux terminal, use the following command:

```
quartus_pgm –c byteblasterII –m jtag –o bpv\;design.pof ←
```

Related Information

- About Quartus Prime Scripting

The jtagconfig Debugging Tool

You can use the `jtagconfig` command-line utility (which is similar to the auto detect operation in the Quartus Prime Programmer) to check the devices in a JTAG chain and the user-defined devices.

For more information about the `jtagconfig` utility, type one of the following commands at the command prompt:

```
jtagconfig –h ←

jtagconfig --help ←
```

Note: The help switch does not reference the `-n` switch. The `jtagconfig -n` command shows each node for each JTAG device.

Related Information

Command-Line Scripting

Generating .pmsf using a .msf and a .sof

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>
```

Document Revision History

Table 14-6: Document Revision History

Date	Version	Chages
2015.11.02	15.1.0	Changed instances of *Quartus II* to *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 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.
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 “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.

Date	Version	Chages
May 2011	11.0.0	• Added links to 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 Quartus Prime Help. • Reorganized chapter.
July 2010	10.0.0	• Added links to 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 Information

Quartus Handbook Archive

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.

Aldec Active-HDL and Riviera-PRO Support 15

2015.11.02

QPP5V3 Subscribe Send Feedback

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

Quick Start Example (Active-HDL VHDL)

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

1. Type the following to specify your EDA simulator and executable path in the Quartus Prime software:

   `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.
   - Use Quartus Prime Simulation Library Compiler to automatically compile all required simulation models for your design.
   - Compile Altera simulation models manually:

   ```
   vlib <library1> <altera_library1>
   vcom -strict93 -dbg -work <library1> <lib1_component/pack.vhd> <lib1.vhd>
   ```
3. Create and open the workspace:

   ```
   createdesign <workspace name> <workspace path>
   opendesign -a <workspace name>.adf
   ```
4. Create the work library and compile the netlist and testbench files:

   ```
   vlib work
   vcom  -strict93  -dbg -work work <output netlist> <testbench file>
   ```
5. Load the design:

   ```
   vsim +access+r -t 1ps +transport_int_delays +transport_path_delays \
   -L work -L <lib1> -L <lib2> work.<testbench module name>
   ```
6. Run the simulation in the Active-HDL simulator.

Aldec Active-HDL and Riviera-PRO Guidelines

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

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.

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.

Table 15-1: 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
```

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.

Before you begin

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 Quartus Prime Settings File (**.qsf**).

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

Using Simulation Setup Scripts

The Quartus Prime software can generate a 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.

Document Revision History

Table 15-2: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *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 Information

Quartus Handbook Archive

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.

Synopsys VCS and VCS MX Support 16

2015.11.02

QPP5V3 Subscribe Send Feedback

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

Quick Start Example (VCS with Verilog)

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

1. Type the following to specify your EDA simulator and executable path in the Quartus Prime software:

   `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.
   - Use Quartus Prime Simulation Library Compiler to automatically compile all required simulation models for your design.
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
```

VCS and QuestaSim Guidelines

The following guidelines apply to simulation of Altera 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.

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.

Table 16-1: 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 <Altera device family \
library>.v +transport_int_delays +pulse_int_e/0 +pulse_int_r/0 \
+transport_path_delays +pulse_e/0 +pulse_r/0
```

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.

Before you begin

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 Quartus Prime Settings File (**.qsf**).

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

Generating Power Analysis Files

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

Before you begin

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

1. In the 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.

VCS Simulation Setup Script Example

The 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 16-1: Using Command-line to Pass Simulation Variables

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

Example 16-2: 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 16-3: 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>
```

Document Revision History

Table 16-2: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *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 Information

Quartus Handbook Archive

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.

Mentor Graphics ModelSim and QuestaSim Support 17

2015.11.02

QPP5V3

Subscribe Send Feedback

You can integrate a supported EDA simulator into the Quartus Prime design flow. This document provides guidelines for simulation of designs with Mentor Graphics® ModelSim-Altera®, ModelSim, or QuestaSim software. Altera provides the entry-level ModelSim-Altera software, along with precompiled Altera simulation libraries, to simplify simulation of Altera designs.

Note: The latest version of the ModelSim-Altera 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-Altera software to simulate Verilog HDL modules and IP cores. Alternatively, you can purchase separate co-simulation software.

Related Information

Simulating Altera Designs on page 1-1

Managing Quartus Prime Projects

Quick Start Example (ModelSim with Verilog)

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

1. Type the following to specify your EDA simulator and executable path in the Quartus Prime software:

   `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.
- Use Quartus Prime Simulation Library Compiler to automatically compile all required simulation models for your design.
- Type the following commands to create and map Altera simulation libraries manually, and then compile the models manually:

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

3. Compile your design and testbench files:

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

4. Load the design:

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

ModelSim, ModelSim-Altera, and QuestaSim Guidelines

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

Using ModelSim-Altera Precompiled Libraries

Precompiled libraries for both functional and gate-level simulations are provided for the ModelSim-Altera 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 <*ModelSim-Altera path*>**/altera/** must be compatible with the version of the Quartus Prime software that creates the simulation netlist. To verify compatibility of precompiled libraries with your version of the Quartus Prime software, refer to the <*ModelSim-Altera path*>**/altera/version.txt** file. This file indicates the Quartus Prime software version and build of the precompiled libraries.

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

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.

Before you begin

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 Quartus Prime Settings File (**.qsf**).

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

Passing Parameter Information from Verilog HDL to VHDL

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

Before you begin

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 Quartus Prime Settings File (**.qsf**).

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

Example 17-1: 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.

Increasing Simulation Speed

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

Before you begin

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-Altera simulator.

```
vlog -fast -05
```

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.

Table 17-1: 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-Altera 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
```

Viewing Error 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.
```

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 Quartus Prime software. You can then run the script from the ModelSim, QuestaSim, or ModelSim-Altera software to generate a timing <*filename*>.vcd. for use in the Quartus Prime PowerPlay power analyzer.

Before you begin

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

1. In the 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. Click **Tools** > **Run EDA Simulation** > **EDA Gate Level Simulation**. 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), and all other files for simulation. ModelSim then automatically runs the generated **.do** to start the simulation.
6. Stop the simulation if your testbench does not have a break point. ModelSim generates the .vcd only after simulation ends with the End Simulation function.

Viewing Simulation Waveforms

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

Before you begin

To view a waveform from a **.wlf** through ModelSim-Altera, ModelSim, or QuestaSim, perform the following steps:

1. Type `vsim` at the command line. The **ModelSim/QuestaSim** or **ModelSim-Altera** 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-Altera, 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.
   You can convert your **.wlf** to a **.vcd** by using the `wlf2vcd` command

Simulating with ModelSim-Altera Waveform Editor

You can use the ModelSim-Altera 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-Altera. With the ModelSim-Altera waveform editor, you can create and edit waveforms, drive simulation directly from created waveforms, and save created waveforms into a stimulus file.

Related Information

ModelSim Web Page

ModelSim Simulation Setup Script Example

The 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.

Unsupported Features

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

- Altera does not support companion licensing for ModelSim AE.
- The USB software guard is not supported by versions earlier than Mentor Graphics ModelSim software version 5.8d.
- For ModelSim-Altera 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 Information

- **ModelSim-Altera Software Web Page**

Document Revision History

Table 17-2: Document Revision History

Date	Version	Changes
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-Altera software.
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 Information

Quartus Handbook Archive

For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.

Cadence Incisive Enterprise (IES) Support 18

2015.11.02

QPP5V3 Subscribe Send Feedback

You can integrate your supported EDA simulator into the *Quartus Prime*design flow. This chapter provides specific guidelines for simulation of *Quartus Prime* designs with the Cadence Incisive Enterprise (IES) software.

Quick Start Example (NC-Verilog)

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

1. Type the following to specify your EDA simulator and executable path in the Quartus Prime software:

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

   `set_global_assignment -name EDA_SIMULATION_TOOL "NC-Verilog (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.
   - Use Quartus Prime Simulation Library Compiler to automatically compile all required simulation models for your design.
   - Map Altera 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 Altera simulation models manually:

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

3. Elaborate your design and testbench with IES:

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

4. Run the simulation:

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

Cadence Incisive Enterprise (IES) Guidelines

The following guidelines apply to simulation of Altera 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.

Using GUI or Command-Line Interfaces

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

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

Table 18-1: 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.

Elaborating Your Design

The simulator automatically reads the **.sdo** file during elaboration of the 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.

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 18-1: 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";
```

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.

Before you begin

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 Quartus Prime Settings File (**.qsf**).

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

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 18-2: 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

Viewing Simulation Waveforms

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

Before you begin

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.

IES Simulation Setup Script Example

The 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 Qsys 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 18-2: 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
```

Document Revision History

Table 18-3: Document Revision History

Date	Version	Changes
2015.11.02	15.1.0	Changed instances of *Quartus II* to *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 Information

Quartus Handbook Archive
For previous versions of the *Quartus Prime Handbook*, refer to the Quartus Handbook Archive.