Intel® Quartus® Prime Pro Edition User Guide

Debug Tools

Updated for Intel® Quartus® Prime Design Suite: **21.3**

Online Version

Send Feedback

UG-20139

ID: 683819

Version: 2021.10.13

Contents

Send Feedback

1. System Debugging Tools Overview

This chapter provides a quick overview of the tools available in the Intel® Quartus® Prime system debugging suite and discusses the criteria for selecting the best tool for your debug requirements.

1.1. System Debugging Tools Portfolio

The Intel Quartus Prime software provides a portfolio of system debugging tools for real-time verification of your design.

System debugging tools provide visibility by routing (or “tapping”) signals in your design to debugging logic. The Compiler includes the debugging logic in your design and generates programming files that you download into the FPGA or CPLD for analysis.

Each tool in the system debugging portfolio uses a combination of available memory, logic, and routing resources to assist in the debugging process. Because different designs have different constraints and requirements, you can choose the tool that matches the specific requirements for your design, such as the number of spare pins available or the amount of logic or memory resources remaining in the physical device.

1.1.1. System Debugging Tools Comparison

Table 1. System Debugging Tools Portfolio

Tool	Description	Typical Usage
System Console and Debugging Toolkits	• Provides real-time in-system debugging capabilities using available debugging toolkits. • Allows you to read from and write to memory mapped components in a system without a processor or additional software. • Communicates with hardware modules in a design through a Tcl interpreter. • Allows you to take advantage of all the features of the Tcl scripting language. • Supports JTAG and TCP/IP connectivity.	• Perform system-level debugging. • Debug or optimize signal integrity of a board layout even before finishing the design. • Debug external memory interfaces. • Debug an Ethernet Intel FPGA IP interface in real time. • Debug a PCI Express* link at the Physical, Data Link, and Transaction layers. • Debug and optimize high-speed serial links in your board design.
Signal Tap logic analyzer	• Uses FPGA resources. • Samples test nodes, and outputs the information to the Intel Quartus Prime software for display and analysis.	You have spare on-chip memory and you want functional verification of a design running in hardware.
Signal Probe	Incrementally routes internal signals to I/O pins while preserving results from the last place-and-routed design.	You have spare I/O pins and you want to check the operation of a small set of control pins using either an external logic analyzer or an oscilloscope.

continued...

Tool	Description	Typical Usage
Logic Analyzer Interface (LAI)	• Multiplexes a larger set of signals to a smaller number of spare I/O pins. • Allows you to select which signals switch onto the I/O pins over a JTAG connection.	You have limited on-chip memory and a large set of internal data buses to verify using an external logic analyzer. Logic analyzer vendors, such as Tektronics* and Agilent*, provide integration with the tool to improve usability.
In-System Sources and Probes	Provides an easy way to drive and sample logic values to and from internal nodes using the JTAG interface.	You want to prototype the FPGA design using a front panel with virtual buttons.
In-System Memory Content Editor	Displays and allows you to edit on-chip memory.	You want 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	Allows you to communicate with the JTAG interface so that you can develop custom applications.	You want to communicate with custom signals in your design.

Refer to the following for more information about launching and using the available debugging toolkits:

- Launching a Toolkit in System Console on page 172
- Available System Debugging Toolkits on page 174

1.1.2. Suggested Tools for Common Debugging Requirements

Table 2. Tools for Common Debugging Requirements[(1)]

Requirement	Signal Probe	Logic Analyzer Interface (LAI)	Signal Tap Logic Analyzer	Description
More Data Storage	N/A	X	—	An external logic analyzer with the LAI tool allows you to store more captured data than the Signal Tap logic analyzer, because the external logic analyzer can provide access to a bigger buffer. The Signal Probe tool does not capture or store data.
Faster Debugging	X	X	—	You can use the LAI or the Signal Probe tool with external equipment, such as oscilloscopes and mixed signal oscilloscopes (MSOs). This ability provides access to timing mode, which allows you to debug combined streams of data.
Minimal Effect on Logic Design	X	X[(2)]	X[(2)]	The Signal Probe tool incrementally routes nodes to pins, with no effect on the design logic. The LAI adds minimal logic to a design, requiring fewer device resources. The Signal Tap logic analyzer has little effect on the design, because the Compiler considers the debug logic as a separate design partition.
Short Compile and Recompile Time	X	X[(2)]	X[(2)]	Signal Probe uses incremental routing to attach signals to previously reserved pins. This feature allows you to quickly recompile when you change the selection of source signals. The Signal Tap logic analyzer and the LAI can refit their own design partitions to decrease recompilation time.
Sophisticated Triggering Capability	N/A	N/A	X	The triggering capabilities of the Signal Tap logic analyzer are comparable to commercial logic analyzers.

continued...

Requirement	Signal Probe	Logic Analyzer Interface (LAI)	Signal Tap Logic Analyzer	Description
Low I/O Usage	—	—	X	The Signal Tap logic analyzer does not require additional output pins. Both the LAI and Signal Probe require I/O pin assignments.
Fast Data Acquisition	N/A	—	X	The Signal Tap logic analyzer can acquire data at speeds of over 200 MHz. Signal integrity issues limit acquisition speed for external logic analyzers that use the LAI.
No JTAG Connection Required	X	—	—	Signal Probe does not require a host for debugging purposes. The Signal Tap logic analyzer and the LAI require an active JTAG connection to a host running the Intel Quartus Prime software.
No External Equipment Required	—	—	X	The Signal Tap logic analyzer only requires a JTAG connection from a host running the Intel Quartus Prime software or the stand-alone Signal Tap logic analyzer. Signal Probe and the LAI require the use of external debugging equipment, such as multimeters, oscilloscopes, or logic analyzers.

Notes to Table:

1. - X indicates the recommended tools for the feature.
   - — indicates that while the tool is available for that feature, that tool might not give the best results.
   - N/A indicates that the feature is not applicable for the selected tool.

1.1.3. Debugging Ecosystem

The Intel Quartus Prime software allows you to use the debugging tools in tandem to exercise and analyze the logic under test and maximize closure.

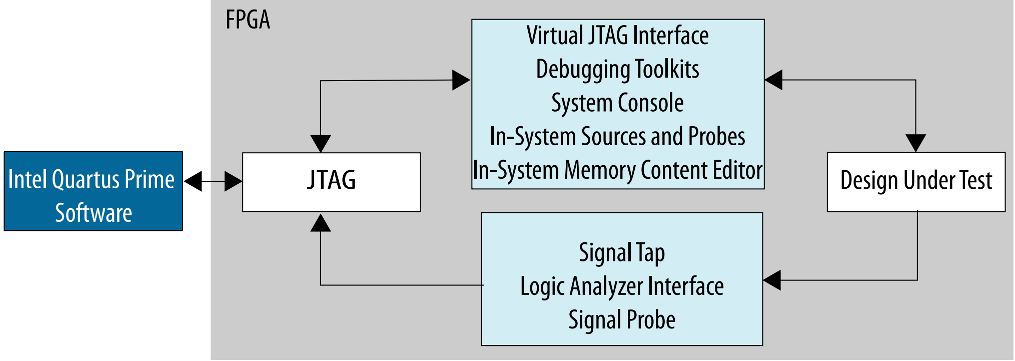
A very important distinction in the system debugging tools is how they interact with the design. All debugging tools in the Intel Quartus Prime software allow you to read the information from the design node, but only a subset allow you to input data at runtime:

Table 3. Classification of System Debugging Tools

Debugging Tool	Read Data from Design	Input Values into the Design	Comments
Signal Tap logic analyzer,	Yes	No	General purpose troubleshooting tools optimized for probing signals in a register transfer level (RTL) netlist
Logic Analyzer Interface			
Signal Probe			
In-System Sources and Probes	Yes	Yes	These tools allow to: • Read data from breakpoints that you define • Input values into your design during runtime
Virtual JTAG Interface			
System Console			
Debugging Toolkits			
In-System Memory Content Editor			

Taken together, the set of on-chip debugging tools form a debugging ecosystem. The set of tools can generate a stimulus to and solicit a response from the logic under test, providing a complete solution.

Figure 1. Debugging Ecosystem at Runtime



1.2. Tools for Monitoring RTL Nodes

The Signal Tap logic analyzer, Signal Probe, and LAI tools are useful for probing and debugging RTL signals at system speed. These general-purpose analysis tools enable you to tap and analyze any routable node from the FPGA.

- In cases when the design has spare logic and memory resources, the Signal Tap logic analyzer can providing fast functional verification of the 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 Signal Probe tools simplify monitoring internal design signals using external equipment.

Related Information

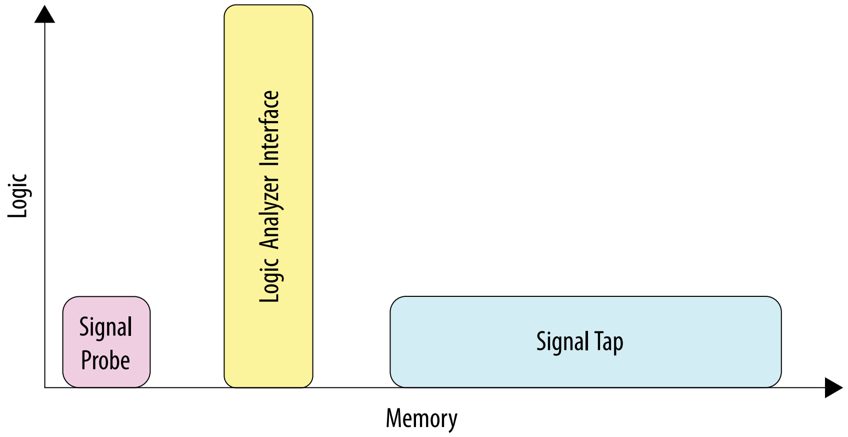
- Quick Design Verification with Signal Probe on page 122
- Design Debugging with the Signal Tap Logic Analyzer on page 28
- In-System Debugging Using External Logic Analyzers on page 127

1.2.1. Resource Usage

The most important selection criteria for these three tools are the remaining resources on the device after implementing the design and the number of spare pins.

Evaluate debugging options early on in the design planning process to ensure that you support the appropriate options in the board, Intel Quartus Prime project, and design. Planning early can reduce debugging time, and eliminates last minute changes to accommodate debug methodologies.

Figure 2. Resource Usage per Debugging Tool



1.2.1.1. Overhead Logic

Any debugging tool that requires a JTAG connection requires SLD infrastructure logic for communication with the JTAG interface and arbitration between instantiated debugging modules. This overhead logic uses around 200 logic elements (LEs), a small fraction of the resources available in any of the supported devices. All available debugging modules in your design share the overhead logic. Both the Signal Tap logic analyzer and the LAI use a JTAG connection.

1.2.1.1.1. For Signal Tap Logic Analyzer

The Signal Tap logic analyzer requires both logic and memory resources. The number of logic resources used depends on the number of signals tapped and the complexity of the trigger logic. However, the amount of logic resources that the Signal Tap logic analyzer uses is typically a small percentage of most designs.

A baseline configuration consisting of the SLD arbitration logic and a single node with basic triggering logic contains approximately 300 to 400 Logic Elements (LEs). Each additional node you add to the baseline configuration adds about 11 LEs. Compared with logic resources, memory resources are a more important factor to consider for your design. Memory usage can be significant and depends on how you configure your Signal Tap logic analyzer instance to capture data and the sample depth that your design requires for debugging. For the Signal Tap logic analyzer, there is the added benefit of requiring no external equipment, as all of the triggering logic and storage is on the chip.

1.2.1.1.2. For Signal Probe

The resource usage of Signal Probe is minimal. Because Signal Probe does not require a JTAG connection, logic and memory resources are not necessary. Signal Probe only requires resources to route internal signals to a debugging test point.

1.2.1.1.3. 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.

1.2.2. Pin Usage

1.2.2.1. For Signal Tap Logic Analyzer

Other than the JTAG test pins, the Signal Tap logic analyzer uses no additional pins. All data is buffered using on-chip memory and communicated to the Signal Tap logic analyzer GUI via the JTAG test port.

1.2.2.2. For Signal Probe

The ratio of the number of pins used to the number of signals tapped for the Signal Probe feature is one-to-one. Because this feature can consume free pins quickly, a typical application for this feature is routing control signals to spare pins for debugging.

1.2.2.3. For Logic Analyzer Interface

The LAI can map up to 256 signals to each debugging pin, depending on available routing resources. The JTAG port controls the active signals mapped to the spare I/O pins. With these characteristics, the LAI is ideal for routing data buses to a set of test pins for analysis.

1.2.3. Usability Enhancements

The Signal Tap logic analyzer, Signal Probe, and LAI tools can be added to your existing design with minimal effects. With the node finder, you can find signals to route to a debugging module without making any changes to your HDL files. Signal Probe inserts signals directly from your post-fit database. The Signal Tap logic analyzer and LAI support inserting signals from both pre-synthesis and post-fit netlists.

1.2.3.1. Incremental Routing

Signal Probe uses the incremental routing feature. The incremental routing feature runs only the Fitter stage of the compilation. This leaves your compiled design untouched, except for the newly routed node or nodes. With Signal Probe, you can save as much as 90% compile time of a full compilation.

1.2.3.2. Automation Via Scripting

As another productivity enhancement, all tools in the on-chip debugging tool set support scripting via the `quartus_stp` Tcl package. For the Signal Tap logic analyzer and the LAI, scripting enables user-defined automation for data collection while debugging in the lab. The System Console includes a full Tcl interpreter for scripting.

1.3. Stimulus-Capable Tools

The In-System Memory Content Editor, In-System Sources and Probes, and Virtual JTAG interface enable you to use the JTAG interface as a general-purpose communication port.

Though you can use all three tools to achieve the same results, there are some considerations that make one tool easier to use in certain applications:

- The In-System Sources and Probes is ideal for toggling control signals.
- The In-System Memory Content Editor is useful for inputting large sets of test data.
- Finally, the Virtual JTAG interface is well suited for advanced users who want to develop custom JTAG solutions.

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 instantiate into your design.

1.3.1. In-System Sources and Probes

In-System Sources and Probes allow you to read and write to a design by accessing JTAG resources.

You instantiate an Intel FPGA IP into your HDL code. This Intel FPGA IP core contains source ports and probe ports that you connect to signals in your design, and abstracts the JTAG interface's transaction details.

In addition, In-System Sources and Probes provide a GUI that displays source and probe ports by instance, and allows you to read from probe ports and drive to source ports. These features make this tool ideal for toggling a set of control signals during the debugging process.

Related Information

Design Debugging Using In-System Sources and Probes on page 144

1.3.1.1. Push Button Functionality

During the development phase of a project, you can debug your design using the In-System Sources and Probes GUI instead of push buttons and LEDs. Furthermore, In-System Sources and Probes supports a set of scripting commands for reading and writing using the Signal Tap logic analyzer. You can also build your own Tk graphical interfaces using the Toolkit API. This feature is ideal for building a virtual front panel during the prototyping phase of the design.

Related Information

Signal Tap Scripting Support on page 116

1.3.2. In-System Memory Content Editor

The In-System Memory Content Editor allows you to quickly view and modify memory content either through a GUI interface or through Tcl scripting commands. The In-System Memory Content Editor works by turning single-port RAM blocks into dual-port RAM blocks. One port is connected to your clock domain and data signals, and the other port is connected to the JTAG clock and data signals for editing or viewing.

Related Information

In-System Modification of Memory and Constants on page 135

1.3.2.1. Generate Test Vectors

Because you can modify a large set of data easily, a useful application for the In-System Memory Content Editor is to generate test vectors for your design. For example, you can instantiate a free memory block, connect the output ports to the logic under test (using the same clock as your logic under test on the system side), and create the glue logic for the address generation and control of the memory. At runtime, you can modify the contents of the memory using either a script or the In-System Memory Content Editor GUI and perform a burst transaction of the data contents in the modified RAM block synchronous to the logic being tested.

1.3.3. System Console

System Console is a framework that you can launch from the Intel Quartus Prime software to start services for performing various debugging tasks. System Console provides you with Tcl scripts and a GUI to access the Platform Designer system integration tool to perform low-level hardware debugging of your design, as well as identify a module by its path, and open and close a connection to a Platform Designer 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.

Related Information

Analyzing and Debugging Designs with System Console on page 158

1.3.3.1. Test Signal Integrity

System Console also allows you to access commands that allow you to control how you generate test patterns, as well as verify the accuracy of data generated by test patterns. You can use JTAG debug commands in System Console to verify the functionality and signal integrity of your JTAG chain. You can test clock and reset signals.

1.3.3.2. Board Bring-Up and Verification

You can use System Console to access programmable logic devices on your development board, perform board bring-up, and perform verification. You can also access software running on a Nios II or Intel FPGA SoC processor, as well as access modules that produce or consume a stream of bytes.

1.3.3.3. Debug with Available Toolkits

System Console provides the hardware debugging infrastructure to run the debugging toolkits that you can enable by the use of debug-enabled Intel FPGA IP. The debugging toolkits can help you to debug external memory interfaces, Ethernet interfaces, PCI Express links, Serial Lite IV links, and high-speed serial links by providing real-time monitoring and debugging of the design running on a board.

Refer to the following for more information about launching and using the available debugging toolkits:

- Launching a Toolkit in System Console on page 172
- Available System Debugging Toolkits on page 174

1.4. Virtual JTAG Interface Intel FPGA IP

The Virtual JTAG Interface Intel FPGA IP provides the finest level of granularity for manipulating the JTAG resource. This Intel FPGA IP 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.

Related Information

- Virtual JTAG (altera_virtual_jtag) IP Core User Guide
- Virtual JTAG Interface (VJI) Intel FPGA IP
  In *Intel Quartus Prime Help*

1.5. System-Level Debug Fabric

During compilation, the Intel Quartus Prime generates the JTAG Hub to allow multiple instances of debugging tools in a design.

Most Intel FPGA on-chip debugging tools use the JTAG port to control and read-back data from debugging logic and signals under test. The JTAG Hub manages the sharing of JTAG resources.

Note: For System Console, you explicitly insert debug IP cores into the design to enable debugging.

The JTAG Hub appears in the project's design hierarchy as a partition named `auto_fab_`*`<number>`*.

1.6. SLD JTAG Bridge

The SLD JTAG Bridge extends the debug fabric across partitions, allowing a higher-level partition (static region or root partition) to access debug signals in a lower-level partition (partial reconfiguration region or core partition).

This bridge consists of two IP components:

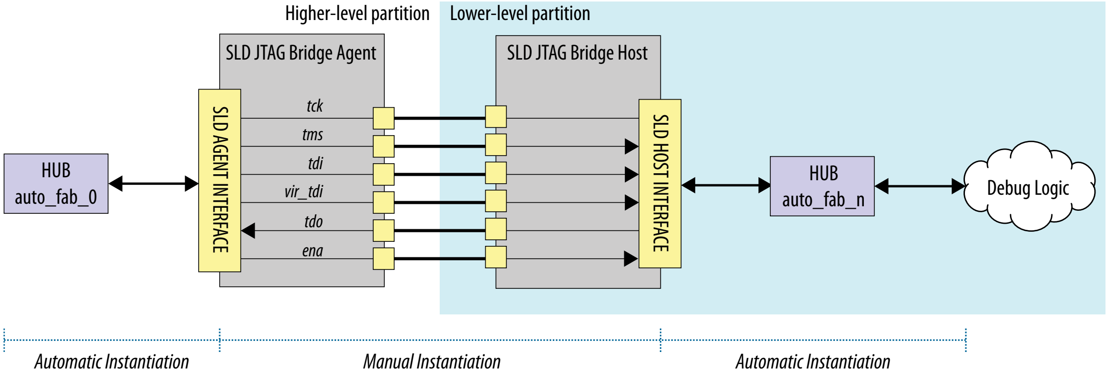
- **SLD JTAG Bridge Agent Intel FPGA IP**—Resides in the higher-level partition.

  Extends the JTAG debug fabric from a higher-level partition to a lower-lever partition containing the SLD JTAG Bridge Host IP. You instantiate the SLD JTAG Bridge Agent IP in the higher-level partition.

- **SLD JTAG Bridge Host Intel FPGA IP**—resides in the lower-level partition. Connects to the PR JTAG hub on one end, and to the SLD JTAG Bridge Agent on the higher-level partition.

  Connects the JTAG debug fabric in a lower-level to a higher-level partition containing the SLD JTAG Bridge Agent IP. You instantiate the SLD JTAG Bridge Host IP in the lower-level partition.

Figure 3. Signals in a SLD JTAG Bridge



For each PR region or reserved core partition you debug, you must instantiate one SLD JTAG Bridge Agent in the higher-level partition and one SLD JTAG Bridge Host in the lower-level partition.

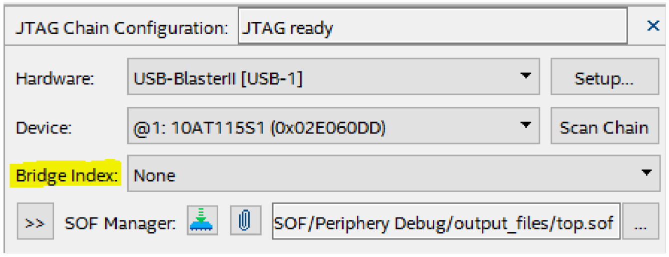
1.6.1. SLD JTAG Bridge Index

The SLD JTAG Bridge Index uniquely identifies instances of the SLD JTAG Bridge present in a design. You can find information regarding the Bridge Index in the synthesis report.

The Intel Quartus Prime software supports multiple instances of the SLD JTAG Bridge in designs. The Compiler assigns an index number to distinguish each instance. The bridge index for the root partition is always None.

When configuring the Signal Tap logic analyzer for the root partition, set the **Bridge Index** value to **None** in the JTAG Chain Configuration window.

Figure 4. JTAG Chain Configuration Bridge Index



JTAG Chain Configuration: JTAG ready

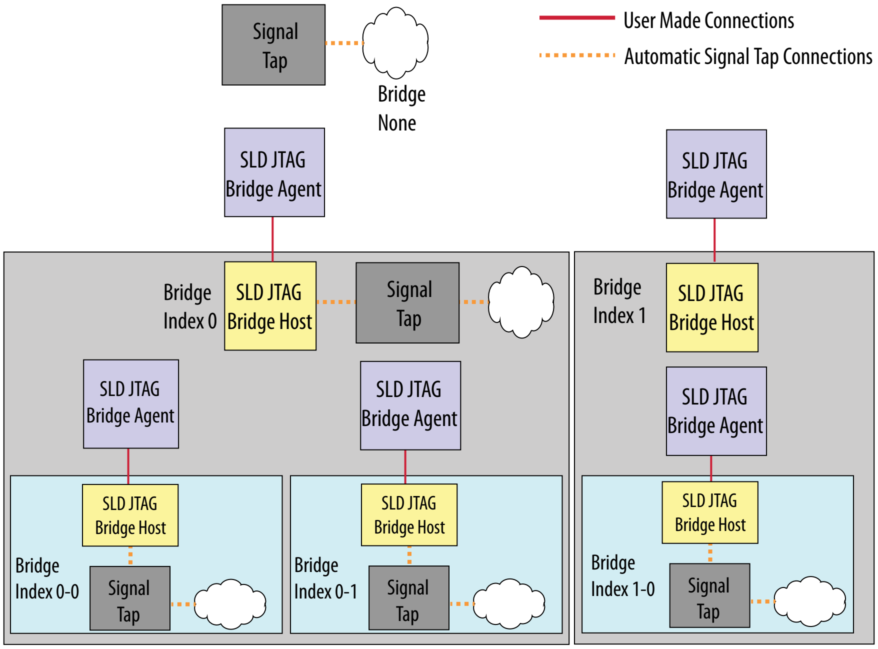
Hardware: USB-BlasterII [USB-1]

Device: @1: 10AT115S1 (0x02E060DD)

Bridge Index: None

SOF Manager: SOF/Periphery Debug/output_files/top.sof

Figure 5. Design with Multiple SLD JTAG Bridges

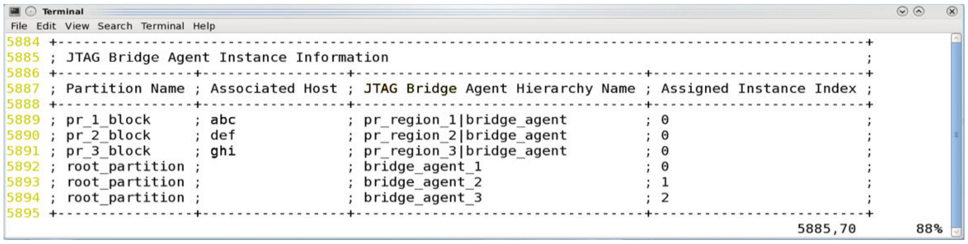


Bridge Index Information in the Compilation Report

Following design synthesis, the Compilation Report lists the index numbers for the SLD JTAG Bridge Agents in the design. Open the **Synthesis ➤ In-System Debugging ➤ JTAG Bridge Instance Agent Information** report for details about how the bridge indexes are enumerated. The reports shows the hierarchy path and the associated index.

In the synthesis report (`<base revision>.syn.rpt`), this information appears in the table **JTAG Bridge Agent Instance Information**.

Figure 6. JTAG Bridge Agent Instance Information



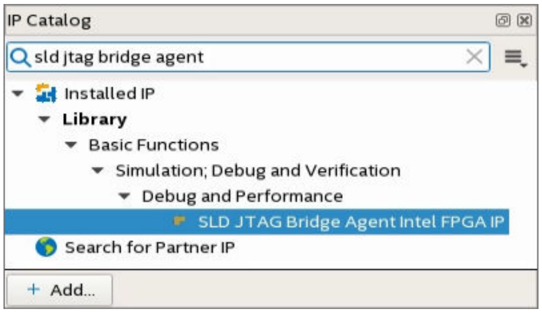
```
+-----------------------------------------------------------------------------------------------+
; JTAG Bridge Agent Instance Information                                                        ;
+----------------+-----------------+----------------------------------+-------------------------+
; Partition Name ; Associated Host ; JTAG Bridge Agent Hierarchy Name ; Assigned Instance Index ;
+----------------+-----------------+----------------------------------+-------------------------+
; pr_1_block     ; abc             ; pr_region_1|bridge_agent         ; 0                       ;
; pr_2_block     ; def             ; pr_region_2|bridge_agent         ; 0                       ;
; pr_3_block     ; ghi             ; pr_region_3|bridge_agent         ; 0                       ;
; root_partition ;                 ; bridge_agent_1                   ; 0                       ;
; root_partition ;                 ; bridge_agent_2                   ; 1                       ;
; root_partition ;                 ; bridge_agent_3                   ; 2                       ;
+----------------+-----------------+----------------------------------+-------------------------+
                                                                                  5885,70     88%
```

1.6.2. Instantiating the SLD JTAG Bridge Agent

To generate and instantiate the SLD JTAG Bridge Agent Intel FPGA IP:

1. On the IP Catalog (**Tools ➤ IP Catalog**), type `SLD JTAG Bridge Agent`.

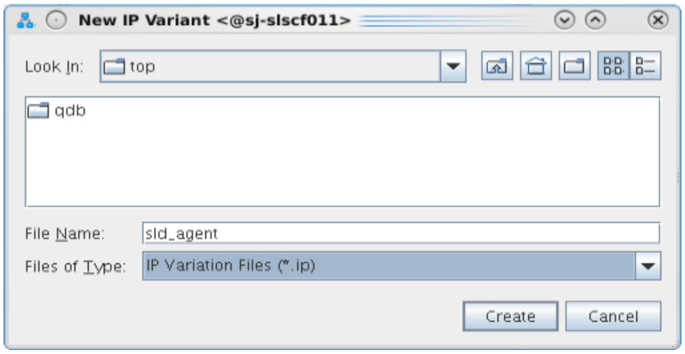
Figure 7. Find in IP Catalog



sld jtag bridge agent

2. Double click **SLD JTAG Bridge Agent Intel FPGA IP**.
3. In the **Create IP Variant** dialog box, type a file name, and then click **Create**.

Figure 8. Create IP Variant Dialog Box



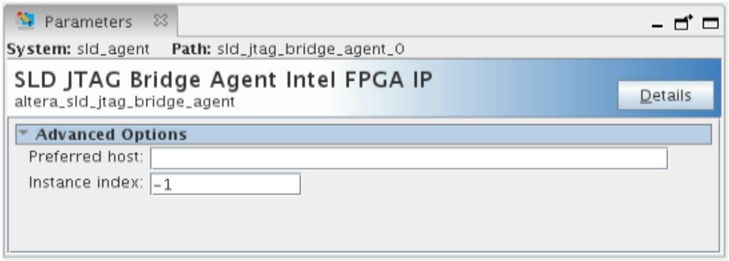
Look In: top

File Name: sld_agent

Files of Type: IP Variation Files (*.ip)

The **IP Parameter Editor Pro** window shows the IP parameters. In most cases, you do not need to change the default values.

Figure 9. SLD JTAG Bridge Agent Intel FPGA IP Parameters


Parameters
SLD JTAG Bridge Agent Intel FPGA IP
altera_sld_jtag_bridge_agent
Details
Advanced Options

System: sld_agent **Path:** sld_jtag_bridge_agent_0

Preferred host:

Instance index: -1

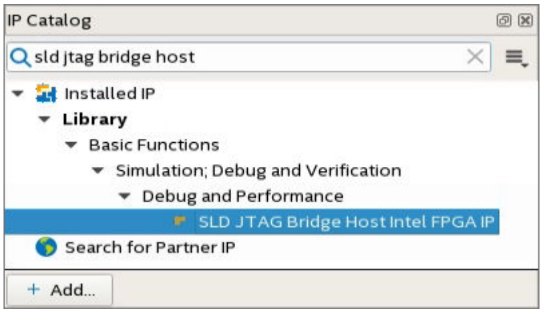
4. Click **Generate HDL**.
5. When the generation completes successfully, click **Close**.
6. If you want an instantiation template, click **Generate ➤ Show Instantiation Template** in the **IP Parameter Editor Pro**.

1.6.3. Instantiating the SLD JTAG Bridge Host

To generate and instantiate the SLD JTAG Bridge Host Intel FPGA IP:

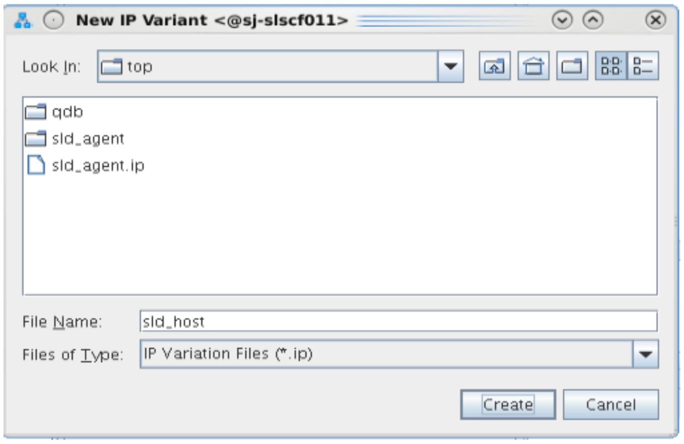
1. On the IP Catalog (**Tools ➤ IP Catalog**), type `SLD JTAG Bridge Host`.

Figure 10. Find in IP Catalog



2. Double click **SLD JTAG Bridge Host Intel FPGA IP**.
3. In the **Create IP Variant** dialog box, type a file name, and then click **Create**.

Figure 11. Create IP Variant Dialog Box



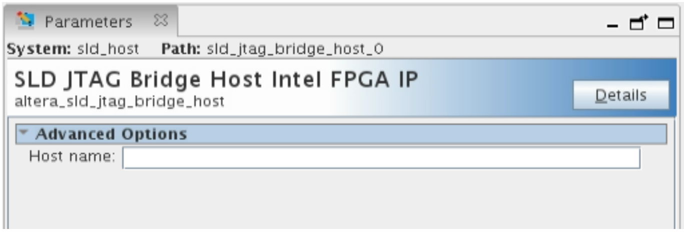
Look In: top

File Name: sld_host

Files of Type: IP Variation Files (*.ip)

The **IP Parameter Editor Pro** window shows the IP parameters. In most cases, you do not need to change the default values.

Figure 12. SLD JTAG Bridge Host Intel FPGA IP Parameters



Parameters
SLD JTAG Bridge Host Intel FPGA IP
altera_sld_jtag_bridge_host
Details
Advanced Options

System: sld_host Path: sld_jtag_bridge_host_0

Host name:

4. Click **Generate HDL**.
5. When the generation completes successfully, click **Close**.
6. If you want an instantiation template, click **Generate ➤ Show Instantiation Template** in the **IP Parameter Editor Pro**.

1.7. Partial Reconfiguration Design Debugging

The following Intel FPGA IP cores support system-level debugging in the static region of a PR design:

- In-System Memory Content Editor
- In-System Sources and Probes Editor
- Virtual JTAG
- Nios II JTAG Debug Module
- Signal Tap Logic Analyzer

In addition, the Signal Tap logic analyzer allows you to debug the static or partial reconfiguration (PR) regions of the design. If you only want to debug the static region, you can use the In-System Sources and Probes Editor, In-System Memory Content Editor, or System Console with a JTAG Avalon bridge.

Related Information

Debugging Partial Reconfiguration Designs with Signal Tap on page 101

1.7.1. Debug Fabric for Partial Reconfiguration Designs

You must prepare the design for PR debug during the early planning stage, to ensure that you can debug the static as well as PR region.

On designs with Partial Reconfiguration, the Compiler generates centralized debug managers—or hubs—for each region (static and PR) that contains system level debug agents. Each hub handles the debug agents in its partition. In the design hierarchy, the hub corresponding to the static region is `auto_fab_0`.

To connect the hubs on parent and child partitions, you must instantiate one SLD JTAG Bridge for each PR region that you want to debug.

Related Information

- PR Design Setup for Signal Tap Debug on page 102
- Debugging Partial Reconfiguration Designs with Signal Tap on page 101

1.7.1.1. Generation of PR Debug Infrastructure

During compilation, the synthesis engine performs the following functions:

- Generates a main JTAG hub in the static region.
- If the static region contains Signal Tap instances, connects those instances to the main JTAG hub.
- Detects bridge agent and bridge host instances.
- Connects the SLD JTAG bridge agent instances to the main JTAG hub.
- For each bridge host instance in a PR region that contains a Signal Tap instance:
  - Generates a PR JTAG hub in the PR region.
  - Connects all Signal Tap instances in the PR region to the PR JTAG hub.
  - Detects instance of the SLD JTAG bridge host.
  - Connects the PR JTAG hub to the JTAG bridge host.

1.8. Preserving Signals for Debugging

The Intel Quartus Prime Pro Edition software allows you to mark and preserve specific signals through the compilation process, which enables visibility of any node within the available system debugging tools.

To ensure that specific nodes in your RTL are available for debugging after the Compiler's synthesis and place-and-route stages, you can apply the `preserve_for_debug` attribute to the signals of interest in your RTL, and also apply the **Enable preserve for debug assignments** project-level `.qsf` assignment.

This section refers to the following terms to explain use of the preserve for debug feature:

Table 4. Debug Signal Preservation Terminology

Term	Description
node	A signal name present in your design RTL and possibly in the compilation netlist for the current project. Typically, the node name refers to the output of a logical unit, such as gate, register, LUT, embedded memory, DSP, or others. The Intel Quartus Prime GUI can display this node name in various locations, such as the Node Finder, when debugging the signals in your design. You can search for this node name to apply constraints and use in debugging operations.
hpath	The Intel Quartus Prime-style hierarchical path, with instance names separated by "|", for example:`foo|boo|node`

1.8.1. Preserve for Debug Overview

The preserve for debug feature allows you to designate nodes in your design for full debugging visibility. In this context, full visibility means that you can ensure that the node name remains in the post-fit netlist generated by Place and Route, with the same name and functionality the design files define.

After you apply preserve for debug, you can easily access these nodes through the Node Finder filters available in the Intel Quartus Prime debugging tools.

Typically, you lose some visibility into the design when you debug using a post-fit netlist. This loss occurs because in the post-fit netlist, the design is already mapped to the device architecture, optimized, and retimed. The Place and Route stage often changes or removes the original signal names. Furthermore, there can be slight changes in the behavior in the post-fit netlist because of inverter push back, or because the visible signal shows only partial behavior due to logic duplication.

Preserve for Debug Use Cases

Preserve for debug is primarily for debugging purposes, and is particularly useful in the Signal Tap debugging flow, as Preserving Signals for Monitoring and Debugging on page 39 describes.

In addition, use of preserve for debug can also be helpful in any of the available system debugging tools, or within any instrumentation logic that you use in your design.

Preserve for Debug Hardware Implementation

Applying the preserve for debug feature has the following effects on hardware implementation:

- Prevents the Compiler from optimizing the specified node.
- Results in LCELL module instantiation for the specified node, impacting the overall timing on the node path.

Application of preserve for debug is the hardware equivalent of using all of the following HDL pragmas on the specified node:

Table 5. Combined Attributes

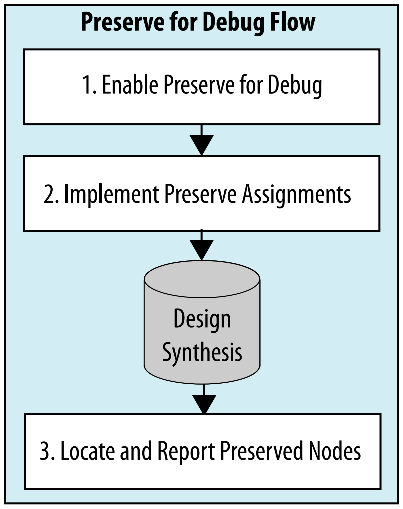
HDL Pragma	Compiler Setting	Description
`preserve`	PRESERVE_REGISTER	Prevents the Compiler from optimizing away or retiming a register.
`keep`	HDL only	Prevents the Compiler from minimizing or removing a particular signal net during combinational logic optimization.
`noprune`	HDL only	Prevents the Compiler from removing or optimizing a fan-out free register.
`dont_merge`	HDL only	Prevents the Compiler from merging a register with a duplicate register.
`dont_replicate`	HDL only	Prevents the Compiler from merging a register with a duplicate register.

1.8.2. Marking Signals for Debug

You can mark (designate) a node for preservation by use of an RTL pragma in your design file, or by specifying an assignment in the project revision `.qsf`.

You can enable or disable preserve for debug at the entity level or globally, so there is no need to individually disable marked signals when ready to compile a production stage design.

Figure 13. Preserve for Debug Flow


Preserve for Debug Flow
1. Enable Preserve for Debug
2. Implement Preserve Assignments
Design Synthesis
3. Locate and Report Preserved Nodes

- Step 1: Enabling Preserve for Debug on page 22
- Step 2: Implement Preserve for Debug Assignments on page 23
- Step 3: Locate and Report Preserve for Debug Nodes on page 24

1.8.2.1. Step 1: Enabling Preserve for Debug

To ensure that the Compiler correctly processes the signals that you mark for preservation, and that the Intel Quartus Prime software Node Finders and filters correctly display these names, you must first turn on the **Enable preserve for debug assignments** setting in the GUI or project revision `.qsf`, as the following methods describe.

Note: The instance-level preserve for debug assignment takes precedence over the global preserve for debug assignment if the two assignments are in opposition to each other (that is, one assignment type is set to On, the other assignment type is set to Off).

Enabling Preserve for Debug In Project Settings on page 22

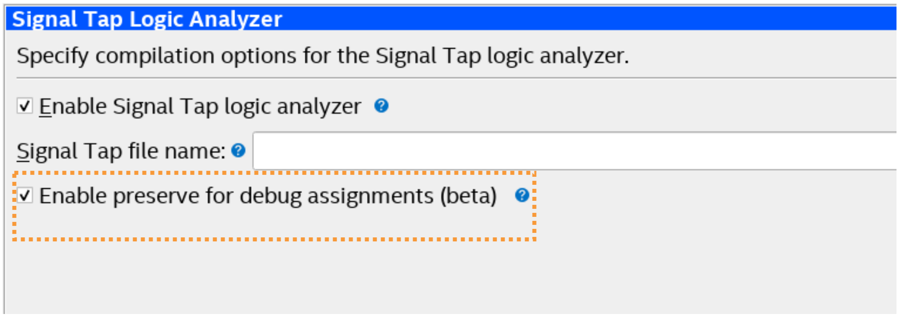
Enabling Preserve for Debug at Instance Level on page 22

1.8.2.1.1. Enabling Preserve for Debug In Project Settings

To enable preserve for debug in the project settings:

1. In the Intel Quartus Prime software, click **Assignments ➤ Settings ➤ Signal Tap Logic Analyzer**.
2. On the **Signal Tap Logic Analyzer** settings page, turn on the **Enable preserve for debug assignments** option. Preserve for debug enables project-wide.

Figure 14. Signal Tap Logic Analyzer Settings



Signal Tap Logic Analyzer

Specify compilation options for the Signal Tap logic analyzer.

☑ Enable Signal Tap logic analyzer

Signal Tap file name:

☑ Enable preserve for debug assignments (beta)

As an alternative to the GUI setting, you can enable or disable project-wide preserve for debug by adding or modifying the following assignment in the project revision `.qsf`:

```
set_global_assignment -name PRESERVE_FOR_DEBUG_ENABLE <ON|OFF>
```

1.8.2.1.2. Enabling Preserve for Debug at Instance Level

You can enable preserve for debug in certain design blocks, and leave the feature disabled in other design blocks.

To enable the assignment at the instance level, you must specify the instance name to enable or disable for preserve for debug, as the following assignment shows:

```
set_instance_assignment -name PRESERVE_FOR_DEBUG_ENABLE ON -to \
     <instance hpath>
```

1.8.2.2. Step 2: Implement Preserve for Debug Assignments

Implement preserve for debug assignments through HDL pragmas in the design files (recommended), or by specifying assignments in the the Assignment Editor or project `.qsf` file directly. The following topics provide more details:

HDL Implementation on page 23

Intel Quartus Prime Settings Implementation on page 23

1.8.2.2.1. HDL Implementation

The recommended method of preserving nodes for debug is to add HDL pragmas or attributes to the design files.

Table 6 on page 23 defines the preserve for debug pragma and `.qsf` assignment setting.

Table 6. Preserve for Debug Pragma

Term	Equivalent (.qsf) Setting	Description
`preserve_for_debug`	PRESERVE_FOR_DEBUG	Prevents the Fitter from optimizing away a register or combinational signal. The pragma also prevents any retiming, merging, and duplication optimization. This optimization prevention applies when the setting, PRESERVE_FOR_DEBUG_ENABLE is ON.

Add HDL pragmas to Verilog HDL design files in the following way:

```
(* preserve_for_debug *) reg my_reg;
```

Add HDL attributes to VHDL design files in the following way:

```
signal keep_wire : std_logic;
attribute keep: boolean;
attribute keep of keep_wire: signal is true;
```

1.8.2.2.2. Intel Quartus Prime Settings Implementation

As an alternative to HDL pragmas, you can specify the following assignment to apply the Preserve for Debug assignment through the `.qsf` settings file directly, or with Assignment Editor.

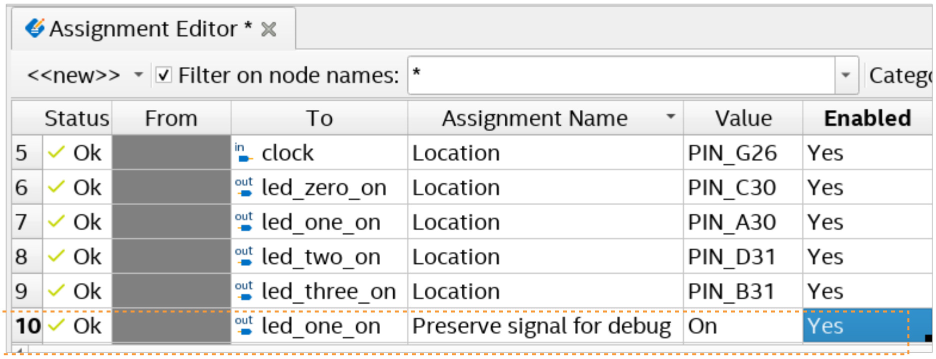
```
set_instance_assignment -name PRESERVE_FOR_DEBUG ON -to \
     <node hpath>
```

Note: This assignment supports the use of wildcards (`*`).

Specifying Preserve Signal for Debug in the Assignment Editor

If you prefer to specify assignments in the Intel Quartus Prime software GUI, rather than in the `.qsf` directly, you can specify the the **Preserve signal for debug** assignment in Assignment Editor (Assignments menu).

Figure 15. Specifying the Preserve Signal for Debug in the Assignment Editor



Assignment Editor *

<<new>> ▾ ☑ Filter on node names: * ▾ Categ

	Status	From	To	Assignment Name	Value	Enabled
5	✓ Ok		in clock	Location	PIN_G26	Yes
6	✓ Ok		out led_zero_on	Location	PIN_C30	Yes
7	✓ Ok		out led_one_on	Location	PIN_A30	Yes
8	✓ Ok		out led_two_on	Location	PIN_D31	Yes
9	✓ Ok		out led_three_on	Location	PIN_B31	Yes
10	✓ Ok		out led_one_on	Preserve signal for debug	On	Yes

1.8.2.3. Step 3: Locate and Report Preserve for Debug Nodes

After running design synthesis, you can locate preserve for debug nodes using the Node Finder in the system debugging tools. In addition, you can view data about the preserve for debug nodes in the Compilation Report. The following topics describe locating and reporting on preserve for debug nodes:

Locating Preserve for Debug Nodes on page 24

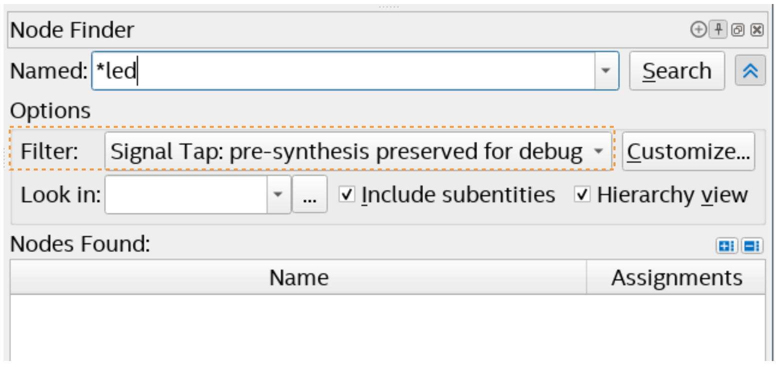
Reporting Preserve for Debug Nodes on page 25

1.8.2.3.1. Locating Preserve for Debug Nodes

The Node Finder includes the following filters that simplify the process of locating the preserve for debug nodes in your project database:

- **Signal Tap: pre-synthesis preserved for debug** filter—shows preserved nodes from the pre-synthesis netlist that generates during Analysis & Elaboration.
- **Signal Tap: post-fitting preserved for debug filter**—shows preserved nodes from the post-fit netlist.

Figure 16. Node Finder with Preserve for Debug Filter



Node Finder

Named: *led ▾ Search

Options

Filter: Signal Tap: pre-synthesis preserved for debug ▾ Customize...

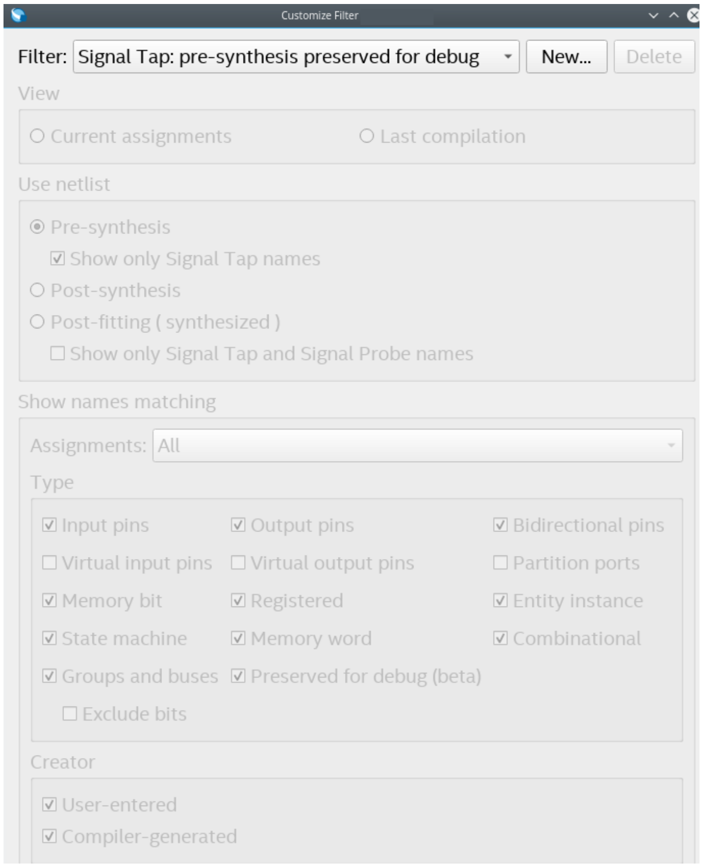
Look in: ▾ ... ☑ Include subentities ☑ Hierarchy view

Nodes Found:

Name	Assignments

You can click the **Customize** button to view the Node Finder search filter settings.

Figure 17. Node Finder Search Filter Settings



Customize Filter

Filter: Signal Tap: pre-synthesis preserved for debug [New...] [Delete]

View

○ Current assignments ○ Last compilation

Use netlist

● Pre-synthesis

☑ Show only Signal Tap names

○ Post-synthesis

○ Post-fitting (synthesized)

□ Show only Signal Tap and Signal Probe names

Show names matching

Assignments: All

Type

☑ Input pins ☑ Output pins ☑ Bidirectional pins

□ Virtual input pins □ Virtual output pins □ Partition ports

☑ Memory bit ☑ Registered ☑ Entity instance

☑ State machine ☑ Memory word ☑ Combinational

☑ Groups and buses ☑ Preserved for debug (beta)

□ Exclude bits

Creator

☑ User-entered

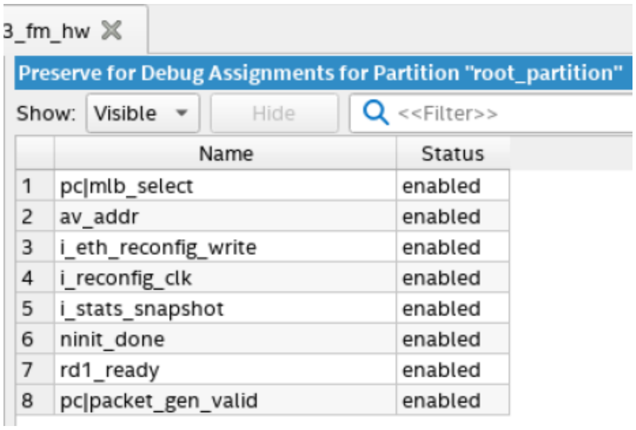
☑ Compiler-generated

1.8.2.3.2. Reporting Preserve for Debug Nodes

You can view data about preserve for debug nodes in the Compilation Report **Preserve for Debug** folder following Analysis & Synthesis.

The Preserve for Debug Assignments for Partition report is located in **Tools ➤ Compilation Report ➤ Synthesis ➤ Partition <name> ➤ Preserve for Debug**.

Figure 18. Preserve for Debug Assignments for Partition Report



Preserve for Debug Assignments for Partition "root_partition"

	Name	Status
1	pc|mlb_select	enabled
2	av_addr	enabled
3	i_eth_reconfig_write	enabled
4	i_reconfig_clk	enabled
5	i_stats_snapshot	enabled
6	ninit_done	enabled
7	rd1_ready	enabled
8	pc|packet_gen_valid	enabled

1.9. System Debugging Tools Overview Revision History

The following revision history applies to this chapter:

Document Version	Intel Quartus Prime Version	Changes
2021.10.04	21.3	• Added new *Preserving Signals for Debugging* section. • Removed obsolete *Remote Debugging* topic. This feature is not supported in the Intel Quartus Prime Pro Edition software. • Removed obsolete *Remote Debugging* chapter 8. This feature is not supported in the Intel Quartus Prime Pro Edition software.
2020.09.28	20.3	• Revised "System Debugging Tools Comparison" to reflect replacement of Transceiver Toolkit with the available debugging toolkits. • Revised "Debugging Ecosystem" to reflect replacement of Transceiver Toolkit with the available debugging toolkits.
2019.09.30	19.3	• Clarified meaning of PR and static regions in "Partial Reconfiguration Design Debugging" topic. • Removed references to Application Notes 693.
2018.09.24	18.1	• Added figures about SLD JTAG Bridge. • Added information about block-based design.
2018.05.07	18.0	• Moved here information about debug fabric on PR designs from the *Design Debugging with the Signal Tap Logic Analyzer* chapter.
2017.05.08	17.0	• Combined Altera JTAG Interface and Required Arbitration Logic topics into a new updated topic named System-Level Debugging Infrastructure. • Added topic: Debug the Partial Reconfiguration Design with System Level Debugging Tools.
2016.10.31	16.1	• Implemented Intel rebranding.
2015.11.02	15.1	Changed instances of *Quartus II* to *Intel Quartus Prime*.
June 2014	14.0	Added information that System Console supports the Tk toolkit.
November 2013	13.1	Dita conversion. Added link to Remote Debugging over TCP/IP for Altera SoC Application Note.
June 2012	12.0	Maintenance release.

continued...

Document Version	Intel Quartus Prime Version	Changes
November 2011	10.0	Maintenance release. Changed to new document template.
December 2010	10.0	Maintenance release. Changed to new document template.
July 2010	10.0	Initial release

2. Design Debugging with the Signal Tap Logic Analyzer

2.1. Signal Tap Logic Analyzer Introduction

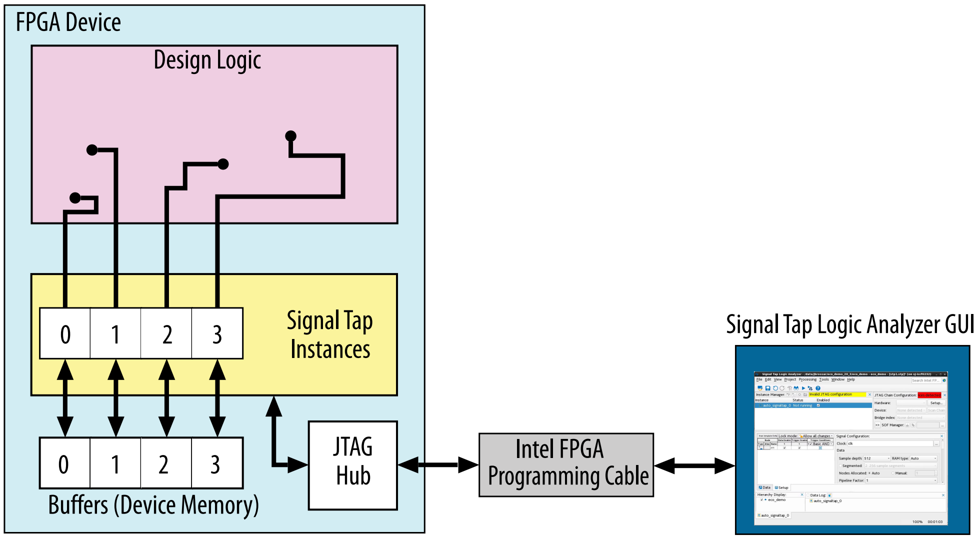
The Signal Tap logic analyzer, available in the Intel Quartus Prime software, captures and displays the real-time signal behavior in an Intel FPGA design. Use the Signal Tap logic analyzer to probe and debug the behavior of internal signals during normal device operation, without requiring extra I/O pins or external lab equipment.

By default, the Signal Tap logic analyzer captures data continuously from the signals you specify while the logic analyzer is running. To capture and store only specific signal data, you specify conditions that *trigger* the start or stop of data capture. A trigger activates when the trigger conditions are met, stopping analysis and displaying the data. You can save the captured data in device memory for later analysis, and filter data that is not relevant.

Signal Tap Logic Analyzer *Instance*

You enable the logic analyzer functionality by defining one or more instances of the Signal Tap logic analyzer in your project. You can define the properties of the Signal Tap instance in the Signal Tap logic analyzer GUI, or by HDL instantiation of the Signal Tap Logic Analyzer Intel FPGA IP. After design compilation, you configure the target device with your design (including any Signal Tap instances), which enables data capture and communication with the Signal Tap logic analyzer GUI over a JTAG connection.

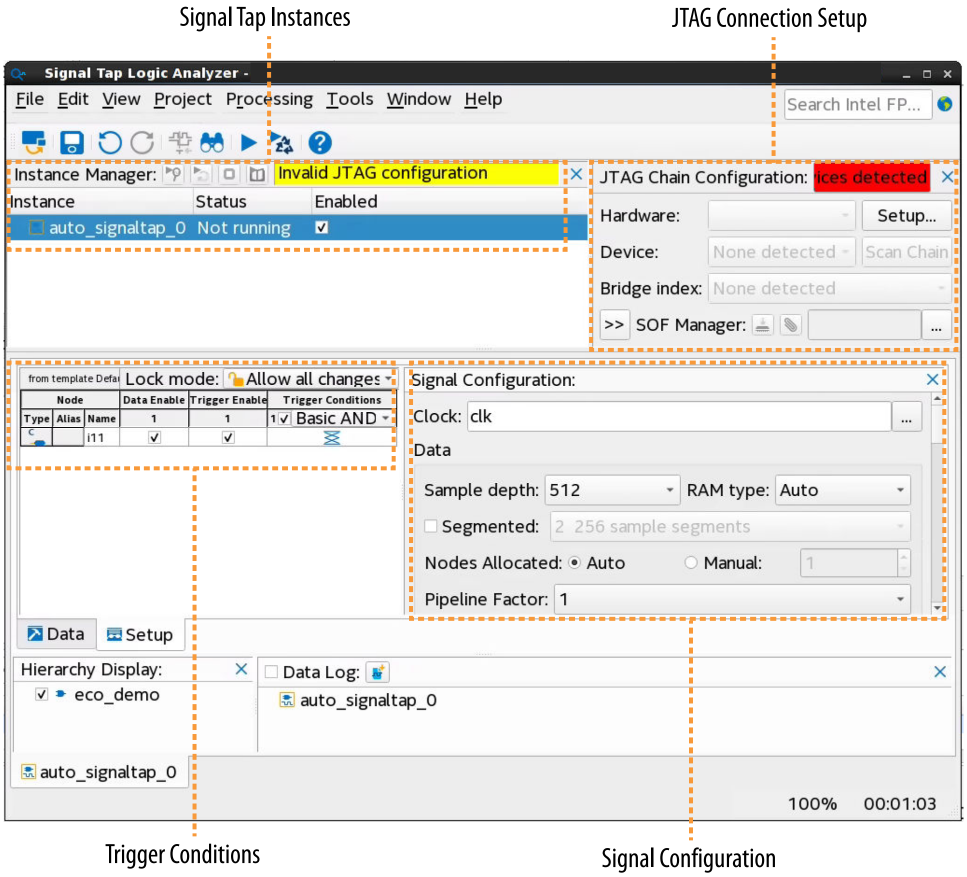
Figure 19. Signal Tap Logic Analyzer Block Diagram



Signal Tap Logic Analyzer *GUI*

The Signal Tap logic analyzer GUI helps you to rapidly define and modify Signal Tap signal configuration and JTAG connection settings, displays the captured signals during analysis, starts and stops analysis, and displays and records signal data. When you configure a Signal Tap instance in the GUI, Signal Tap preserves the instance settings in a Signal Tap Logic Analyzer file (`.stp`) for reuse.

Figure 20. Signal Tap Logic Analyzer GUI



Signal Tap Instances

JTAG Connection Setup

Trigger Conditions

Signal Configuration

Instance Manager: Invalid JTAG configuration

Instance	Status	Enabled
auto_signaltap_0	Not running	☑

JTAG Chain Configuration: ices detected

Hardware:

Device: None detected

Bridge index: None detected

SOF Manager:

Lock mode: Allow all changes

Signal Configuration:

Clock: clk

Data

Sample depth: 512 RAM type: Auto

□ Segmented: 2 256 sample segments

Nodes Allocated: ● Auto ○ Manual: 1

Pipeline Factor: 1

Hierarchy Display: ☑ eco_demo

Data Log: auto_signaltap_0

Signal Tap Logic Analyzer and Simulator Integration

You can integrate the Signal Tap logic analyzer with your supported simulator environment. Signal Tap can readily generate a list of "simulator-aware" nodes to tap for any design hierarchy. Tapping this set of nodes then provides full visibility into the entire design hierarchy for direct observation of all internal signal states in your RTL simulator.

Signal Tap also supports automatic RTL simulation testbench creation, allowing you to export acquired Signal Tap hardware data directly into your RTL simulator and observe signals beyond those that you specify for tapping in Signal Tap. You can produce simulation events using the live data traffic to replicate in your simulator.

Signal Tap Logic Analyzer Capabilities

The Signal Tap logic analyzer supports a high number of channels, a large sample depth, fast clock speeds, and other features described in the *Key Signal Tap Logic Analyzer Capabilities* table.

Table 7. Key Signal Tap Logic Analyzer Capabilities

Capability	Benefit
Multiple logic analyzers in a single device, or in multiple devices in a single chain	Capture data from multiple clock domains and from multiple devices at the same time.
Up to 10 trigger conditions for each analyzer instance	Send complex data capture commands to the logic analyzer for greater accuracy and problem isolation.
Power-up trigger	Capture signal data for triggers that occur after device programming, but before manually starting the logic analyzer.
Custom trigger HDL object	Define a custom trigger in Verilog HDL or VHDL and tap specific instances of modules across the design hierarchy, without manual routing of all the necessary connections.
State-based triggering flow	Organize triggering conditions to precisely define data capture.
Flexible buffer acquisition modes	Precise control of data written into the acquisition buffer. Discard data samples that are not relevant to the debugging of your design.
MATLAB* integration with MEX function	Collect Signal Tap capture data into a MATLAB integer matrix.
RTL simulator integration	Easily create a set of nodes to tap for the design hierarchy, and observe all internal signal states in your RTL simulator. Automatic testbench creation allows you to inject acquired Signal Tap data directly into your RTL simulator.
Up to 4,096 channels per logic analyzer instance	Samples many signals and wide bus structures.
Up to 128K samples per instance	Captures a large sample set for each channel.
Fast clock frequencies	Synchronous sampling of data nodes using the same clock tree driving the logic under test.
Compatible with other debugging utilities	Use the Signal Tap logic analyzer in tandem with any JTAG-based on-chip debugging tool, such as an In-System Memory Content editor, to change signal values in real-time.
Floating-Point Display Format	• Single-precision floating-point format **IEEE754 Single (32-bit)**. • Double-precision floating-point format **IEEE754 Double (64-bit)**.

2.1.1. Signal Tap Hardware and Software Requirements

All editions of the Intel Quartus Prime design software include the Signal Tap logic analyzer GUI and Signal Tap Logic Analyzer Intel FPGA IP. The Signal Tap logic analyzer is also available as a stand-alone application.

During data acquisition, the memory blocks in the FPGA device store the captured data, and then transfer the data to the Signal Tap logic analyzer over a JTAG communication cable, such as Intel FPGA Ethernet Cable or Intel FPGA Download Cable.

The Signal Tap logic analyzer requires the following hardware and software to perform logic analysis:

- The Signal Tap logic analyzer included with the Intel Quartus Prime software, or the Signal Tap logic analyzer standalone software and standalone Programmer software.
- An Intel FPGA download or communications cable.
- An Intel development kit, or your own design board with a JTAG connection to the device under test.

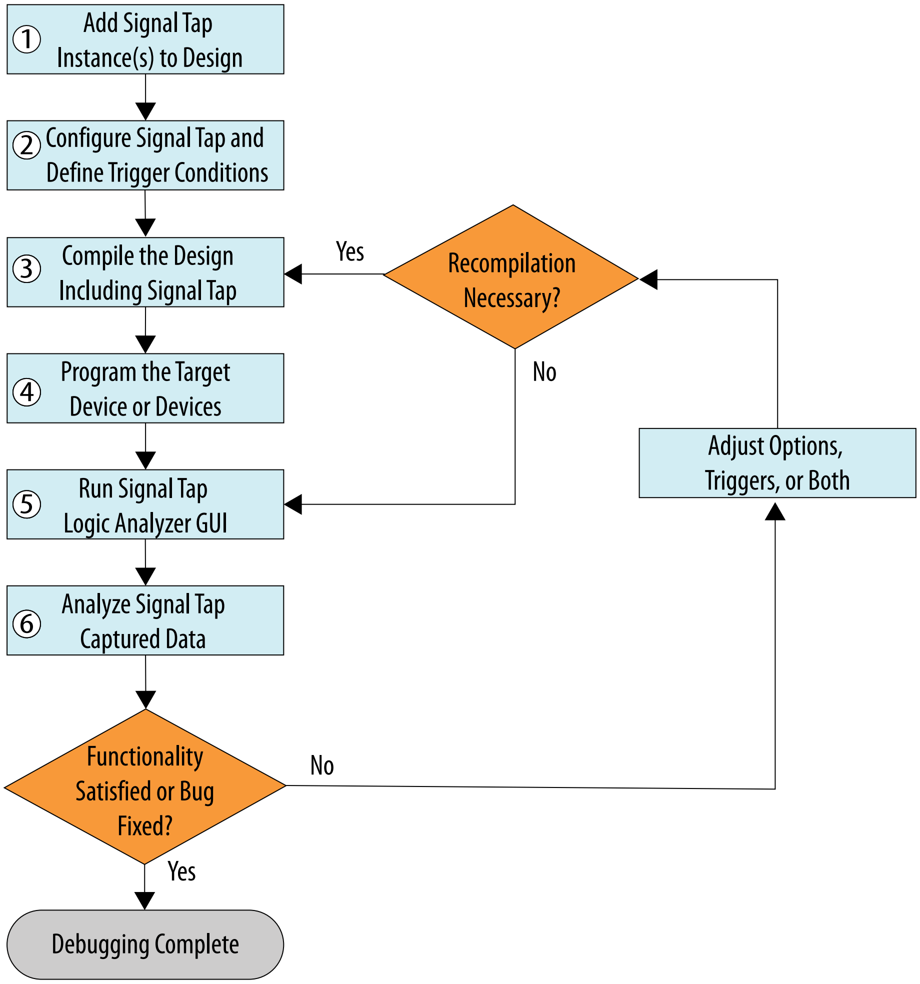
Related Information

Running the Stand-Alone Version of Signal Tap on page 116

2.2. Signal Tap Debugging Flow

To use the Signal Tap logic analyzer to debug your design, you compile your design that includes one or more Signal Tap instances that you define, configure the target device, and then run the logic analyzer to capture and analyze signal data.

Figure 21. Signal Tap Debugging Flow


① Add Signal Tap Instance(s) to Design
② Configure Signal Tap and Define Trigger Conditions
③ Compile the Design Including Signal Tap
④ Program the Target Device or Devices
⑤ Run Signal Tap Logic Analyzer GUI
⑥ Analyze Signal Tap Captured Data
Functionality Satisfied or Bug Fixed?
Yes
Debugging Complete
No
Adjust Options, Triggers, or Both
Recompilation Necessary?
Yes
No

The following steps describe the Signal Tap debugging flow in detail:

- Step 1: Add the Signal Tap Logic Analyzer to the Project on page 33
- Step 2: Configure the Signal Tap Logic Analyzer on page 38
- Step 3: Compile the Design and Signal Tap Instances on page 80
- Step 4: Program the Target Hardware on page 83
- Step 5: Run the Signal Tap Logic Analyzer on page 83
- Step 6: Analyze Signal Tap Captured Data on page 90

2.3. Step 1: Add the Signal Tap Logic Analyzer to the Project

To debug a design using the Signal Tap logic analyzer, you must first define one or more Signal Tap instances and add them to your project. You then compile the Signal Tap instances, along with your design. You can define a Signal Tap instance in the Signal Tap logic analyzer GUI or by HDL instantiation.

To help you get started quickly, the Signal Tap logic analyzer GUI includes preconfigured templates for various trigger conditions and applications. You can then modify the settings the template applies and adjust trigger conditions in the Signal Tap logic analyzer GUI.

Alternatively, you can define a Signal Tap instance by parameterizing an instance of the Signal Tap Logic Analyzer Intel FPGA IP, and then instantiating the Signal Tap entity or module in an HDL design file.

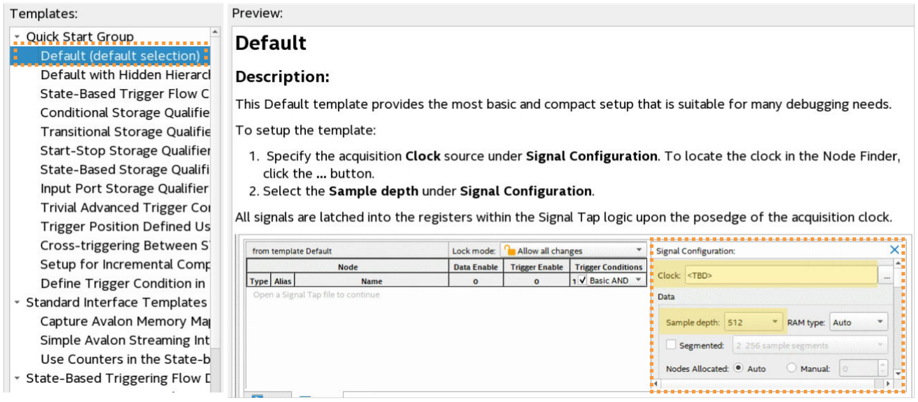
If you want to monitor multiple clock domains simultaneously, you can add additional instances of the logic analyzer to your design, limited only by the available resources in your device.

2.3.1. Creating a Signal Tap Instance with the Signal Tap GUI

When you define one or more Signal Tap instances in the GUI, Signal Tap stores the trigger and signal configuration settings in a Signal Tap Logic Analyzer File (`.stp`). You can open a `.stp` to reload that Signal Tap configuration.

1. Open a project and run **Analysis & Synthesis** on the Compilation Dashboard.
2. To create a Signal Tap instance with the Signal Tap logic analyzer GUI, perform one of the following:
   - Click **Tools ➤ Signal Tap Logic Analyzer**.
   - Click **File ➤ New ➤ Signal Tap Logic Analyzer File**.

Figure 22. Signal Tap file Templates



Templates:

- Quick Start Group
  - Default (default selection)
  - Default with Hidden Hierarcl
  - State-Based Trigger Flow C
  - Conditional Storage Qualifie
  - Transitional Storage Qualifie
  - Start-Stop Storage Qualifier
  - State-Based Storage Qualifi
  - Input Port Storage Qualifier
  - Trivial Advanced Trigger Coı
  - Trigger Position Defined Us
  - Cross-triggering Between S
  - Setup for Incremental Comp
  - Define Trigger Condition in
- Standard Interface Templates
  - Capture Avalon Memory Maı
  - Simple Avalon Streaming Int
  - Use Counters in the State-b
- State-Based Triggering Flow D

Preview:

Default

Description:

This Default template provides the most basic and compact setup that is suitable for many debugging needs.

To setup the template:

1. Specify the acquisition **Clock** source under **Signal Configuration**. To locate the clock in the Node Finder, click the **...** button.
2. Select the **Sample depth** under **Signal Configuration**.

All signals are latched into the registers within the Signal Tap logic upon the posedge of the acquisition clock.

from template Default — Lock mode: Allow all changes

Node		Data Enable	Trigger Enable	Trigger Conditions
Type	Alias	Name	0	0	1 ✓ Basic AND

Open a Signal Tap file to continue

Signal Configuration:

Clock: <TBD>

Data

Sample depth: 512 RAM type: Auto

□ Segmented: 2 256 sample segments

Nodes Allocated: ● Auto ○ Manual: 0

3. Select a Signal Tap file template. The **Preview** describes the setup and **Signal Configuration** the template applies. Refer to Signal Tap File Templates on page 114.

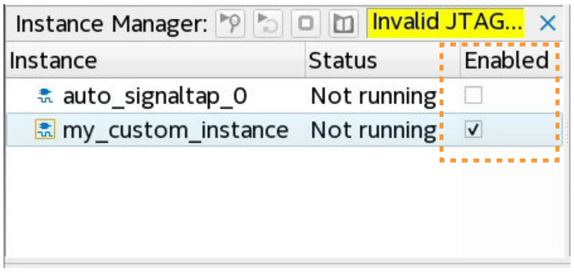
4. Click **Create**. The Signal Tap logic analyzer GUI opens with the template options preset for the Signal Tap instance.
5. Under **Signal Configuration**, specify the acquisition **Clock** and optionally modify other settings, as Step 2: Configure the Signal Tap Logic Analyzer on page 38 describes.
6. When you save or close the Signal Tap instance, click **Yes** when prompted to add the Signal Tap instance to the project.

2.3.1.1. Manging Signal Tap Instances

You can manage the properties of different Signal Tap instances in the **Instance Manager** pane. You can enable or disable one or more instances to specify whether your project includes the instance the next time you run compilation. If you enable or disable instances, you must recompile the design to implement the changes.

The **Instance Manager** toolbar allows you to **Run Analysis** and **Stop Analysis**, or start **Autorun Analysis**, which starts the Signal Tap logic analyzer in a repetitive acquisition mode, providing continuous display update.

Figure 23. Enable and Disable Signal Tap Instances in Instance Manager

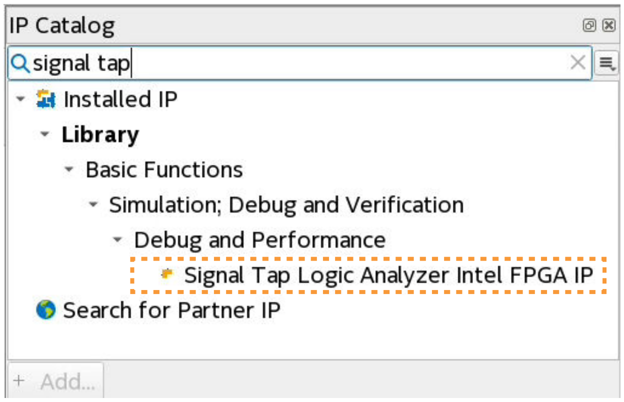


Instance	Status	Enabled
auto_signaltap_0	Not running	□
my_custom_instance	Not running	☑

2.3.2. Creating a Signal Tap Instance by HDL Instantiation

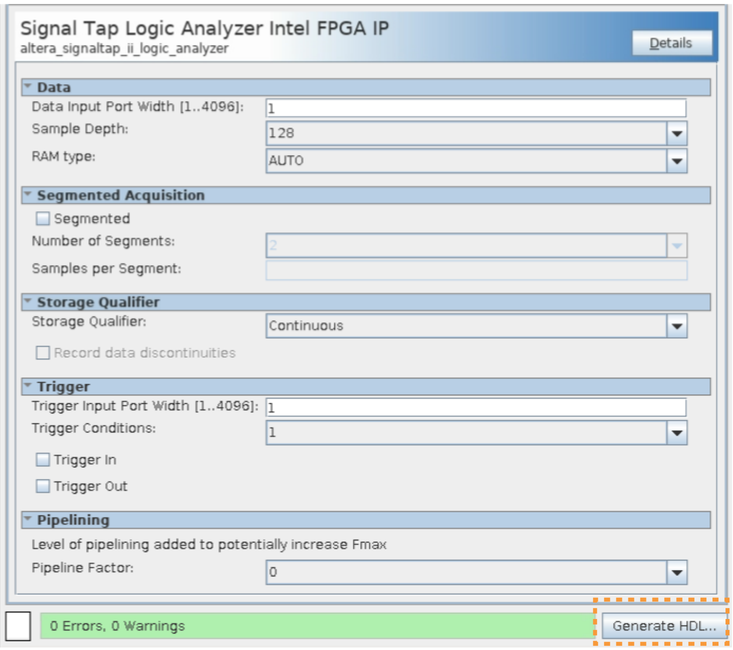
You can create a Signal Tap Instance by HDL instantiation, rather than using the Signal Tap logic analyzer GUI. When you use HDL instantiation, you first parameterize and instantiate the Signal Tap Logic Analyzer Intel FPGA IP in your RTL. Next, you compile the design and IP, and run a Signal Tap analysis using the generated `.stp` file. Follow these steps to create a Signal Tap instance by HDL instantiation:

Figure 24. Signal Tap Logic Analyzer Intel FPGA IP



1. From the Intel Quartus Prime IP Catalog (**View ➤ IP Catalog**), locate and double-click the **Signal Tap Logic Analyzer Intel FPGA IP**.
2. In the **New IP Variant** dialog box, specify the **File Name** for your Signal Tap instance, and then click **Create**. The IP parameter editor displays the available parameter settings for the Signal Tap instance.
3. In the parameter editor, specify the **Data**, **Segmented Acquisition**, **Storage Qualifier**, **Trigger**, and **Pipelining** parameters, as Signal Tap Intel FPGA IP Parameters on page 36 describes.
4. Click Generate HDL. The parameter editor generates the HDL implementation of the Signal Tap instance according your specifications.

Figure 25. IP Parameter Editor



Signal Tap Logic Analyzer Intel FPGA IP
altera_signaltap_ii_logic_analyzer

Data
Data Input Port Width [1..4096]: 1
Sample Depth: 128
RAM type: AUTO

Segmented Acquisition
□ Segmented
Number of Segments: 2
Samples per Segment:

Storage Qualifier
Storage Qualifier: Continuous
□ Record data discontinuities

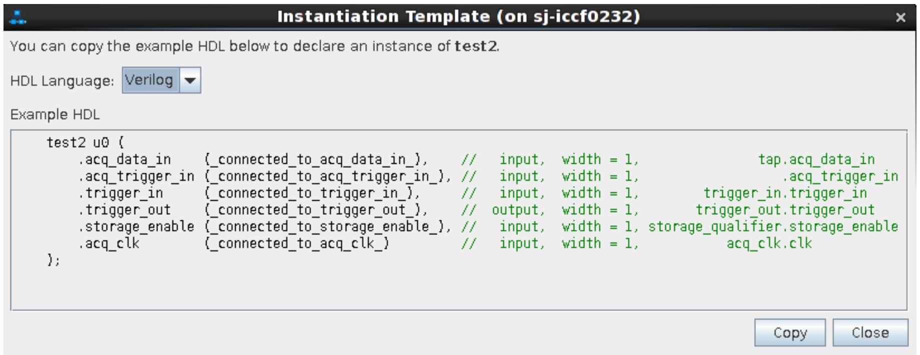
Trigger
Trigger Input Port Width [1..4096]: 1
Trigger Conditions: 1
□ Trigger In
□ Trigger Out

Pipelining
Level of pipelining added to potentially increase Fmax
Pipeline Factor: 0

0 Errors, 0 Warnings

5. To instantiate the Signal Tap instance in your RTL, click **Generate ➤ Show Instantiation Template** in the parameter editor. **Copy** the **Instantiation Template** contents into your RTL.

Figure 26. Signal Tap Logic Analyzer Intel FPGA IP Instantiation Template



Instantiation Template (on sj-iccf0232)

You can copy the example HDL below to declare an instance of **test2**.

HDL Language: Verilog

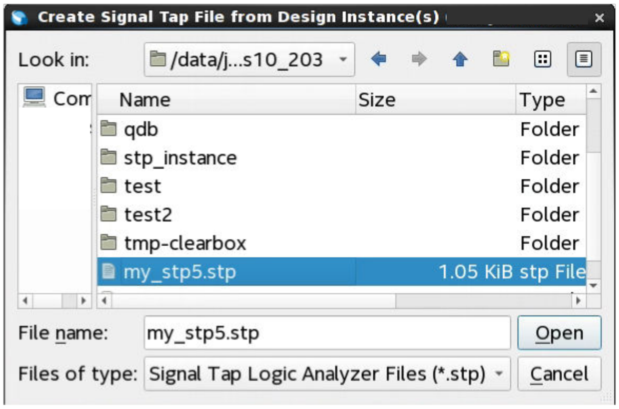
Example HDL

```
    test2 u0 (
        .acq_data_in    (_connected_to_acq_data_in_),    //   input,  width = 1,               tap.acq_data_in
        .acq_trigger_in (_connected_to_acq_trigger_in_), //   input,  width = 1,                  .acq_trigger_in
        .trigger_in     (_connected_to_trigger_in_),     //   input,  width = 1,        trigger_in.trigger_in
        .trigger_out    (_connected_to_trigger_out_),    //  output,  width = 1,       trigger_out.trigger_out
        .storage_enable (_connected_to_storage_enable_), //   input,  width = 1, storage_qualifier.storage_enable
        .acq_clk        (_connected_to_acq_clk_)         //   input,  width = 1,           acq_clk.clk
    );
```

6. Run at least the Analysis & Synthesis stage of the Compiler to synthesize the RTL (including Signal Tap instance) by clicking **Processing ➤ Start ➤ Start Analysis & Synthesis**. Alternatively, you can run full compilation and the Assembler at this point if ready.

7. When the Compiler completes, click **Create/Update ➤ Create Signal Tap File from Design Instance** to create a `.stp` file for analysis in the Signal Tap logic analyzer GUI.

Figure 27. Create Signal Tap File from Design Instances Dialog Box



Create Signal Tap File from Design Instance(s)

Look in: /data/j...s10_203

Name	Size	Type
qdb		Folder
stp_instance		Folder
test		Folder
test2		Folder
tmp-clearbox		Folder
my_stp5.stp	1.05 KiB	stp File

File name: my_stp5.stp

Files of type: Signal Tap Logic Analyzer Files (*.stp)

Note: If your project contains partial reconfiguration partitions, the PR partitions display in a tree view. Select a partition from the view, and click **Create Signal Tap file**. The resulting `.stp` file that generates contains all HDL instances in the corresponding PR partition. The resulting `.stp` file does not include the instances in any nested partial reconfiguration partition.

8. To analyze the Signal Tap instance, click **File ➤ Open** and select the `.stp` file. The Signal Tap instance opens in the Signal Tap logic analyzer GUI for analysis. All the fields are read-only, except runtime-configurable trigger conditions.

9. Modify any runtime-configurable trigger conditions, as Runtime Reconfigurable Options on page 87 describes.

2.3.2.1. Signal Tap Intel FPGA IP Parameters

The Signal Tap Intel FPGA IP has the following parameters:

Table 8. Signal Tap Intel FPGA IP Parameters

Parameter Groups	Parameter Descriptions
Data	• **Data Input Port Width**—from 1 to 4096. Default is 1. • **Sample Depth**—number of samples to collect from 0-128K. Default is 128. • **RAM type**—memory type for sample collection and storage. The **Auto** (default), **M20K/M10K/M9K**, **MLAB/LUTRAM**, and **M144K** options are available.
Segmented Acquisition	Specifies options for organizing the captured data buffer:

continued...

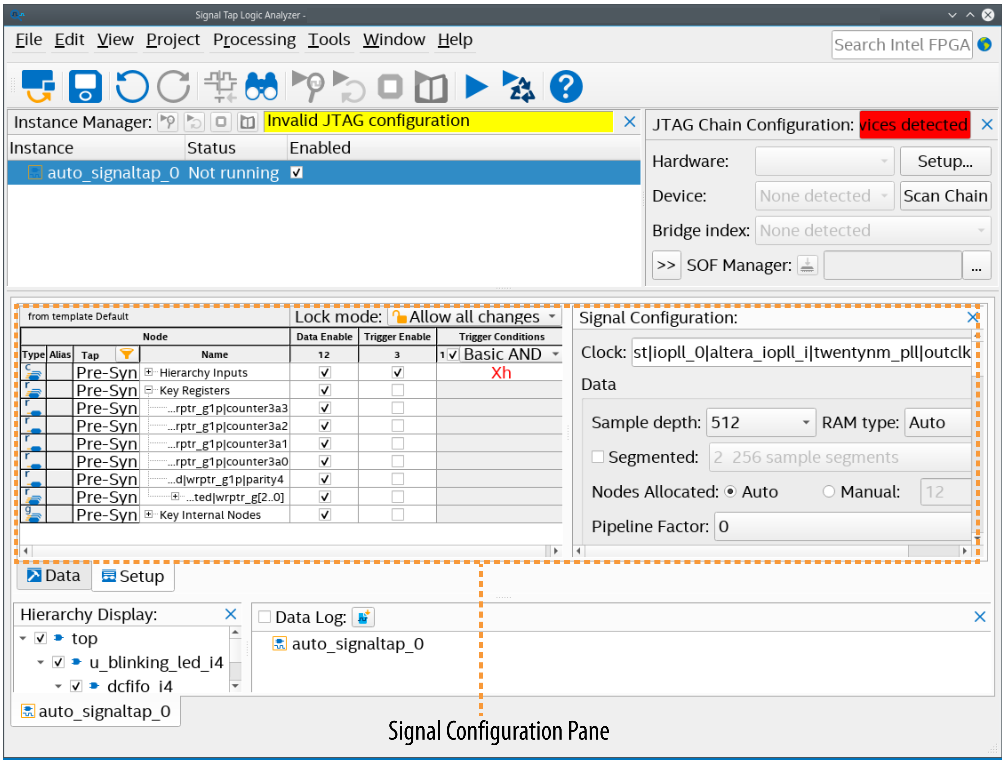
Parameter Groups	Parameter Descriptions
	• **Segmented**—the memory space is split into separate buffers. Each buffer acts as a separate FIFO with its own set of trigger conditions, and behaves as a non-segmented buffer. Only a single buffer is active during an acquisition. Default is off. • **Number of Segments**—specifies the number of segments in each memory space. Default is 2. • **Samples per Segments**—the number of samples Signal Tap captures per segment. Default is 64.
Storage Qualifier	Specifies the **Continuous** or **Input Port** method, and whether to **Record data discontinuities**.
Trigger	• **Trigger Input Port Width**—from 1 to 4096. Default is 1. • **Trigger Conditions**—number of trigger conditions or levels you are implementing 1-10. Default is 1. • **Trigger In**—enables and creates a port for the Trigger In. • **Trigger Out**—enables and creates a port for the Trigger Out.
Pipelining	The Pipeline Factor specifies the levels of pipelining added for potential f_{MAX} improvement from 0 to 5. Default is 0.

2.4. Step 2: Configure the Signal Tap Logic Analyzer

You must configure the Signal Tap logic analyzer before you can capture and analyze data. You can configure instances of the Signal Tap logic analyzer by specifying options in the Signal Tap **Signal Configuration** pane.

When you use the available Signal Tap templates to create a new Signal Tap instance, the template specifies many of the initial option values automatically.

Figure 28. Signal Tap Logic Analyzer Signal Configuration Pane



Signal Tap Logic Analyzer -
File Edit View Project Processing Tools Window Help
Search Intel FPGA

Instance Manager: Invalid JTAG configuration

Instance	Status	Enabled
auto_signaltap_0	Not running	☑

JTAG Chain Configuration: vices detected

Hardware: Setup...

Device: None detected Scan Chain

Bridge index: None detected

>> SOF Manager: ...

from template Default

Lock mode: Allow all changes

Node				Data Enable	Trigger Enable	Trigger Conditions
Type	Alias	Tap	Name	12	3	1 ☑ Basic AND
c		Pre-Syn	⊞ Hierarchy Inputs	☑	☑	Xh
r		Pre-Syn	⊟ Key Registers	☑	☐	
r		Pre-Syn	...rptr_g1p|counter3a3	☑	☐	
r		Pre-Syn	...rptr_g1p|counter3a2	☑	☐	
r		Pre-Syn	...rptr_g1p|counter3a1	☑	☐	
r		Pre-Syn	...rptr_g1p|counter3a0	☑	☐	
r		Pre-Syn	...d|wrptr_g1p|parity4	☑	☐	
r		Pre-Syn	⊞ ...ted|wrptr_g[2..0]	☑	☐	
g		Pre-Syn	⊞ Key Internal Nodes	☑	☐	

Signal Configuration:

Clock: st|iopll_0|altera_iopll_i|twentynm_pll|outclk

Data

Sample depth: 512 RAM type: Auto

☐ Segmented: 2 256 sample segments

Nodes Allocated: ● Auto ○ Manual: 12

Pipeline Factor: 0

Basic configuration of the Signal Tap logic analyzer includes specifying values for the following options:

- Preserving Signals for Monitoring and Debugging on page 39
- Specifying the Clock, Sample Depth, and RAM Type on page 41
- Specifying the Buffer Acquisition Mode on page 42
- Adding Signals to the Signal Tap Logic Analyzer on page 44
- Defining Trigger Conditions on page 50
- Specifying Pipeline Settings on page 72
- Filtering Relevant Samples on page 73
- Managing Multiple Signal Tap Configurations on page 99

Related Information

Preventing Changes that Require Full Recompilation on page 41

2.4.1. Preserving Signals for Monitoring and Debugging

The Compiler optimizes the RTL signals during synthesis and place-and-route. Unless preserved, the signal names in your RTL may not exist in the post-fit netlist after signal optimizations. For example, the compilation process can merge duplicate registers, or add tildes (~) to net names that fan-out from a node.

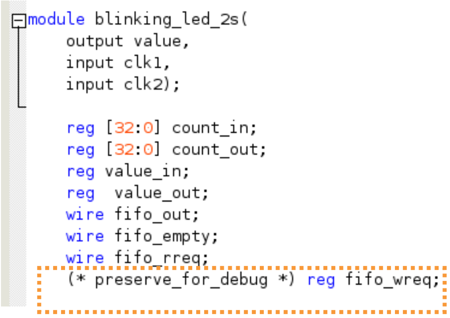
To ensure that specific nodes in your RTL are available for Signal Tap debugging after synthesis and place-and-route, you can apply the `preserve_for_debug` attribute to the signals of interest in your RTL, and also specify the **Enable preserve for debug assignments** project `.qsf` setting. Refer to `.qsf` syntax in Table 9 on page 40.

When you preserve signals using this technique, the Compiler generates the Preserve for Debug Assignments report following synthesis that shows the status and name of all nodes with the `preserve_for_debug` attribute in your RTL.

Follow these steps to preserve signals for monitoring and debugging:

1. In your design RTL, mark signals that you want to preserve with the `preserve_for_debug` attribute:

Figure 29. preserve_for_debug Attribute



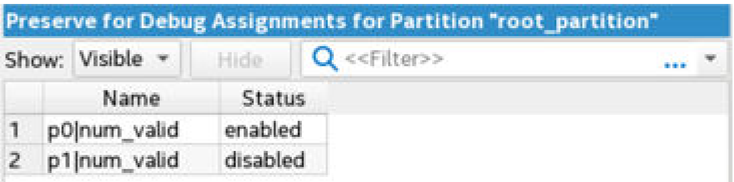
```
module blinking_led_2s(
    output value,
    input clk1,
    input clk2);

    reg [32:0] count_in;
    reg [32:0] count_out;
    reg value_in;
    reg  value_out;
    wire fifo_out;
    wire fifo_empty;
    wire fifo_rreq;
    (* preserve_for_debug *) reg fifo_wreq;
```

2. Open the project containing Signal Tap in the Intel Quartus Prime software and perform one of the following:
   - To enable preservation and reporting for specific instances, click **Assignments ➤ Assignment Editor**, and then specify the **Enable preserve for debug assignments** assignment **To** any instance of interest.

     Or
   - To enable preservation and reporting project-wide, in **Assignments ➤ Settings ➤ Signal Tap Logic Analyzer**, turn on **Enable preserve for debug assignments**.[1]
3. To synthesize the design, on the Compilation Dashboard, click **Analysis & Synthesis**. The Compilation Report appears when synthesis is complete.
4. To view the results of signal preservation, open the Preserve for Debug Assignments report located in the **Synthesis ➤ Partition <name> ➤ Preserve for Debug** report folder.

[1] The global project setting has a more limited impact and does not preserve signals that would otherwise be optimized away in their local context.

Figure 30. Preserve for Debug Assignments Report



Preserve for Debug Assignments for Partition "root_partition"

	Name	Status
1	p0|num_valid	enabled
2	p1|num_valid	disabled

5. Run full compilation to perform place and route of the design and Signal Tap instance, as Step 3: Compile the Design and Signal Tap Instances on page 80 describes. The debug signals that you preserve in step 2 persist through the Fitter into the finalized compilation database.
6. Optionally, make some incremental changes to the Signal Tap configuration without running full recompilation, as Changing the Post-Fit Signal Tap Target Nodes on page 84 describes.

Table 9. Debug Signal Preservation Methods

Method	Description	Example
`preserve_for_debug_enable`	Set this assignment to **On** to preserve any nodes or hierarchies marked with `preserve_for_debug`. If set to **Off** or not used, any `preserve_for_debug` assignments are ignored. Use this as a quick way to disable all debug node preservation when optimizing a completed design. The Compiler reports these nodes in the Preserve for Debug Assignments report following compilation.	`set_instance_assignment -name PRESERVE_FOR_DEBUG_ENABLE ON`
`preserve_for_debug` (**Enable preserve for debug assignments** in the Assignment Editor)	Instance-specific `.qsf` assignment that overrides the global assignment and enables preservation of all types of nodes through synthesis post-synthesis or post-fit debugging purposes. When **On**, this assignment enables preservation for the hierarchy that you specify. You can enable or disable this with the **Preserve signal for debug** assignment in the Assignment Editor. The Compiler reports these nodes in the Preserve for Debug Assignments report following compilation.	`set_instance_assignment -name PRESERVE_FOR_DEBUG ON -to <`*`node hpath`*`>`

Note: For more information about preserving signals, refer to *Preserving Registers During Synthesis*, in the *Intel Hyperflex™ Architecture High-Performance Design Handbook* and *Preserving a System Module, Interface, or Port for Debugging* in the *Intel Quartus Prime Pro Edition User Guide: Platform Designer*.

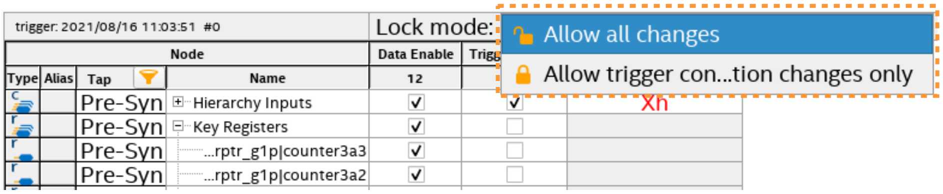
Related Information

- Intel Hyperflex Architecture High-Performance Design Handbook
- Changing the Post-Fit Signal Tap Target Nodes on page 84
- Preserving Signals for Debugging on page 20
- Preserving a System Module, Interface, or Port for Debugging, Intel Quartus Prime Pro Edition User Guide: Platform Designer

2.4.2. Preventing Changes that Require Full Recompilation

Making some types of changes to the Signal Tap configuration require full recompilation to implement. If you want to ensure that you make no changes to the Signal Tap configuration that require full recompilation, select **Allow trigger condition changes only** for the **Lock mode**. Alternatively, you can enable **Allow all changes**, including those changes that require full compilation or recompilation to implement.

Figure 31. Allow Trigger Conditions Change Only



trigger: 2021/08/16 11:03:51 #0

Lock mode:

- Allow all changes
- Allow trigger con...tion changes only

Node				Data Enable	Trigg
Type	Alias	Tap	Name	12	
		Pre-Syn	⊞ Hierarchy Inputs	✓	✓ Xh
		Pre-Syn	⊟ Key Registers	✓	□
		Pre-Syn	...rptr_g1p|counter3a3	✓	□
		Pre-Syn	...rptr_g1p|counter3a2	✓	□

Related Information

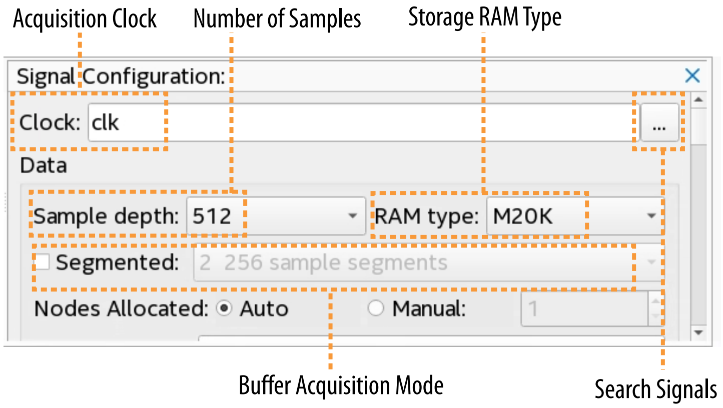
Recompiling Only Signal Tap Changes on page 80

2.4.3. Specifying the Clock, Sample Depth, and RAM Type

You must specify options for the acquisition clock, sample depth, and data storage on the **Signal Configuration** pane before using Signal Tap.

Note: The Signal Tap file templates automatically specify appropriate initial values for some of these options.

Figure 32. Clock, Sample Depth, and Data Storage Options



Acquisition Clock

Number of Samples

Storage RAM Type

Signal Configuration:

Clock: clk ...

Data

Sample depth: 512

RAM type: M20K

□ Segmented: 2 256 sample segments

Nodes Allocated: ● Auto ○ Manual: 1

Buffer Acquisition Mode

Search Signals

Specifying the Acquisition Clock

Signal Tap samples data on each positive (rising) edge of the acquisition clock. Therefore, Signal Tap requires a clock signal from your design to control the logic analyzer data acquisition. For best data acquisition, specify a global, non-gated clock that is synchronous to the signals under test. Refer to the Timing Analysis section of the Compilation Report for the maximum frequency of the logic analyzer clock.

- To specify the acquisition clock signal, enter a signal name from your design for the **Clock** setting in **Single Configuration**.

Note: Consider the following when specifying the acquisition clock:

- If you do not assign an acquisition clock, Signal Tap automatically creates clock pin `auto_stp_external_clk`. You must then make a pin assignment to this signal, and ensure that a clock signal in your design drives the acquisition clock.
- Using a transceiver recovered clock as the acquisition clock can cause incorrect or unexpected behavior, particularly when the transceiver recovered clock is the acquisition clock with the power-up trigger feature.
- Specifying a gated acquisition clock can result in unexpected data that does not accurately reflect the behavior of your design.
- Signal Tap does not support sampling on the negative (falling) clock edge.

Specifying Sample Depth

The sample depth determines the number of samples the logic analyzer captures and stores in the data buffer, for each signal. In cases with limited device memory resources, you can reduce the sample depth to reduce resource usage.

- To specify the sample depth, select the number of samples from the **Sample depth** list under **Single Configuration**. Available sample depth range is from 0 to 128K.

Specifying the RAM Type

You can specify the RAM type and buffer acquisition mode for storage of Signal Tap logic analyzer acquisition data. When you allocate the Signal Tap logic analyzer buffer to a particular RAM block, the entire RAM block becomes a dedicated resource for the logic analyzer.

- To specify the RAM type, select a **Ram type** under **Single Configuration**. Available settings are **Auto**, **MLAB**, or **M20K** RAM.

Use RAM selection to preserve a specific memory block for your design, and allocate another portion of memory for Signal Tap data acquisition. For example, if your design has an application that requires a large block of memory resources, such as a large instruction or data cache, use MLAB blocks for data acquisition and leave M20k blocks for your design.

Related Information

- Adding Nios II Processor Signals with a Plug-In on page 49
- Managing Device I/O Pins, Intel Quartus Prime Pro Edition User Guide: Design Constraints

2.4.4. Specifying the Buffer Acquisition Mode

You can specify how Signal Tap organizes the data capture buffer to potentially reduce the amount of memory that Signal Tap requires for data acquisition.

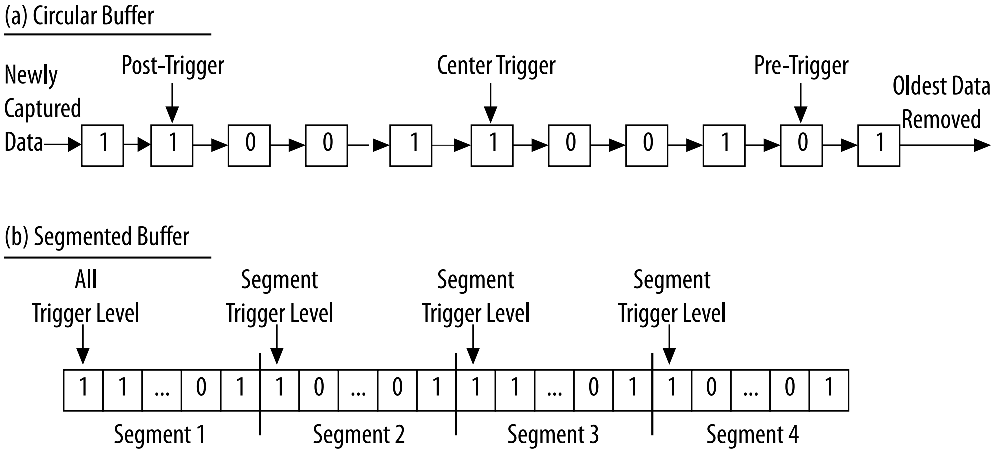
The Signal Tap logic analyzer supports either a non-segmented (or circular) buffer and a segmented buffer.

- **Non-segmented buffer**—the Signal Tap logic analyzer treats the entire memory space as a single FIFO, continuously filling the buffer until the logic analyzer reaches the trigger conditions that you specify.
- **Segmented buffer**—the memory space is split into separate buffers. Each buffer acts as a separate FIFO with its own set of trigger conditions, and behaves as a non-segmented buffer. Only a single buffer is active during an acquisition. The Signal Tap logic analyzer advances to the next segment after the trigger condition or conditions for the active segment has been reached.

When using a non-segmented buffer, you can use the storage qualification feature to determine which samples are written into the acquisition buffer. Both the segmented buffers and the non-segmented buffer with the storage qualification feature help you maximize the use of the available memory space.

Figure 33. Buffer Type Comparison in the Signal Tap Logic Analyzer

The figure illustrates the differences between the two buffer types.



Both non-segmented and segmented buffers can use a preset trigger position (Pre-Trigger, Center Trigger, Post-Trigger). Alternatively, you can define a custom trigger position using the **State-Based Triggering** tab, as Specify Trigger Position on page 58 describes.

2.4.4.1. Non-Segmented Buffer

The non-segmented buffer is the default buffer type in the Signal Tap logic analyzer.

At runtime, the logic analyzer stores data in the buffer until the buffer fills up. From that point on, new data overwrites the oldest data, until a specific trigger event occurs. The amount of data the buffer captures after the trigger event depends on the **Trigger position** setting:

- To capture more data from before the trigger occurs, select **Post trigger position** from the list.
- To capture all data from after the trigger occurs, 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.

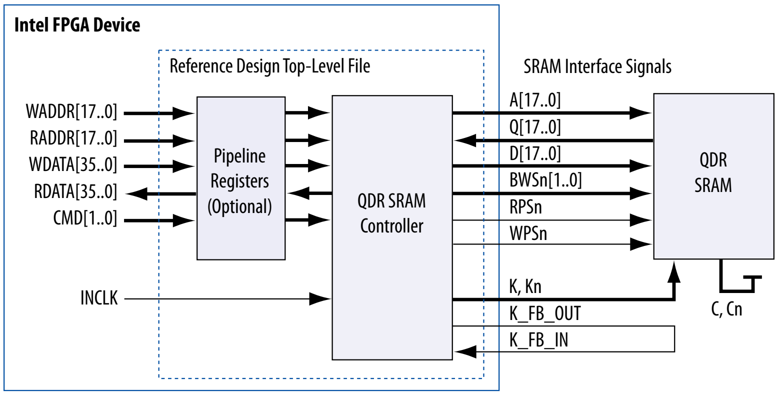
2.4.4.2. Segmented Buffer

In a segmented buffer, the acquisition memory is split into segments of even size, and you define a set of trigger conditions for all segments. Each segment acts as a non-segmented buffer. A segmented buffer allows you to debug systems that contain relatively infrequent recurring events.

If you want to have separate trigger conditions for each of the buffer segments, you must use the state-based trigger flow. The figure shows an example of a segmented buffer system.

Figure 34. System that Generates Recurring Events

In the following example, to ensure that the correct data is written to the SRAM controller, monitor the RDATA port whenever the address `H'0F0F0F0F` is sent into the RADDR port.



The buffer acquisition feature allows you to monitor multiple read transactions from the SRAM device without running the Signal Tap logic analyzer again. You can split the memory to capture the same event multiple times, without wasting allocated memory. The buffer captures as many cycles as the number of segments you define under the **Data** settings in the **Signal Configuration** pane.

To enable and configure buffer acquisition, select **Segmented** in the Signal Tap logic analyzer Editor and choose the number of segments to use. In the example in the figure, selecting 64-sample segments allows you to capture 64 read cycles.

Related Information

Viewing Capture Data Using Segmented Buffers on page 91

2.4.5. Adding Signals to the Signal Tap Logic Analyzer

You add the signals that you want to monitor to the node list in the Signal Tap logic analyzer. You can then select a signals in the node list to define the triggers for the signal.

Adding Pre-Synthesis Signals

You can add expected signals to Signal Tap for monitoring without running synthesis. Pre-synthesis signal names are those names present after Analysis & Elaboration, but before any synthesis optimizations. When you add pre-synthesis signals to Signal Tap for monitoring, you must make all connections to the Signal Tap logic analyzer before running synthesis. The Compiler then automatically allocates the logic and routing resources to make these connections. For signals driving to and from IOEs, pre-synthesis signal names coincide with the pin's signal names.

Refer to Adding Pre-Synthesis or Post-Fit Nodes on page 45.

Adding Simulator-Aware Signals

You can easily generate a list of simulator-aware, pre-synthesis signals to tap for an entire design hierarchy, and then observe all internal signal states in your RTL simulator. This set of simulator-aware nodes can provide full visibility into other untapped nodes in the design hierarchy. You can then export captured Signal Tap signal data data directly into your RTL simulator to observe signal states beyond Signal Tap observability.

Refer to Adding Simulator-Aware Signal Tap Nodes on page 47.

Adding Post-Fit Signals

You can add post-fit signals to Signal Tap for monitoring. Post-fit signal names are those names present in the netlist after physical synthesis optimizations and place-and-route. When you add post-fit signals to Signal Tap for monitoring, you are connecting to actual atoms in the post-fit netlist. You can only monitor signals that exist in the post-fit netlist, and existing routing resources must be available.

In the case of post-fit output signals, monitor the `COMBOUT` or `REGOUT` signal that drives the IOE block. For post-fit input signals, signals driving into the core logic coincide with the pin's signal name.

Note: Because `NOT`-gate push back applies to any register that you monitor, the signal from the atom may be inverted. You can verify the inversion by locating to the signal with the **Locate Node ➤ Locate in Resource Property Editor** or the **Locate Node ➤ Locate in Technology Map Viewer** commands. You can also view post-fit node names in the Resource Property Editor.

Related Information

Signal Tap and Simulator Integration on page 96

2.4.5.1. Adding Pre-Synthesis or Post-Fit Nodes

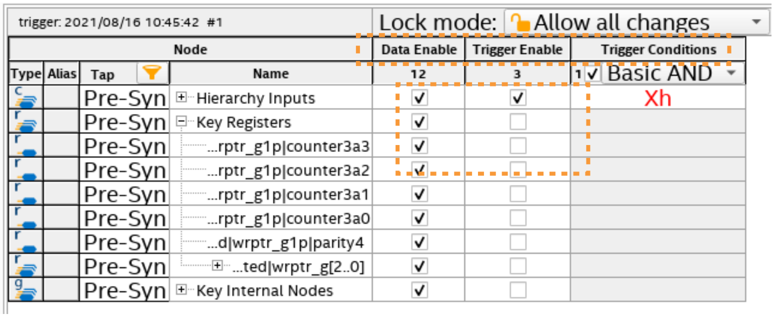
To add one or more pre-synthesis or post-fit signals to the Signal Tap **Node** list for monitoring:

1. Click either of the following commands to generate the pre-synthesis or post-fit design netlist:
   - **Processing ➤ Start ➤ Start Analysis & Elaboration** (generates pre-synthesis netlist)
   - **Processing ➤ Start ➤ Start Fitter** (generates post-fit netlist)
2. In the Signal Tap logic analyzer, Click **Edit ➤ Add Nodes**. The Node Finder appears, allowing you to find and add the signals in your design. The following Filter options are available for finding the nodes you want:

- **Signal Tap: pre-synthesis**—finds signal names present after design elaboration, but before any synthesis optimizations are done. **Signal Tap: pre-synthesis preserved for debug** finds presynthesis signals that you mark with the `preserve_for_debug` pragma, as Preserving Signals for Monitoring and Debugging on page 39 describes.
- **Signal Tap: post-fitting**—finds signal names present after physical synthesis optimizations and place-and-route. **Signal Tap: post-fitting preserved for debug** finds post-fit signals that you mark with the `preserve_for_debug` pragma.

3. In the Node Finder, select one or more nodes that you want to add, and then click the **Copy all to Selected Nodes list** button.
4. Click **Insert**. The nodes are added to the **Setup** tab signal list in the Signal Tap logic analyzer GUI.
5. Specify how the logic analyzer uses the signal by enabling or disabling the **Data Enable**, **Trigger Enable**, or **Storage Enable** option for the signal:
   - **Trigger Enable**—disabling prevents a signal from triggering the analysis, while still showing the signal's captured data.
   - **Data Enable**—disabling prevent capture of data, while still allowing the signal to trigger.

Figure 35. Signal Tap Node List Options for Data Enable and Trigger Enable



trigger: 2021/08/16 10:45:42 #1

Lock mode: Allow all changes

Node				Data Enable	Trigger Enable	Trigger Conditions
Type	Alias	Tap	Name	12	3	1 ✓ Basic AND
c		Pre-Syn	⊞ Hierarchy Inputs	☑	☑	Xh
r		Pre-Syn	⊟ Key Registers	☑	☐	
r		Pre-Syn	...rptr_g1p|counter3a3	☑	☐	
r		Pre-Syn	...rptr_g1p|counter3a2	☑	☐	
r		Pre-Syn	...rptr_g1p|counter3a1	☑	☐	
r		Pre-Syn	...rptr_g1p|counter3a0	☑	☐	
r		Pre-Syn	...d|wrptr_g1p|parity4	☑	☐	
r		Pre-Syn	⊞ ...ted|wrptr_g[2..0]	☑	☐	
g		Pre-Syn	⊞ Key Internal Nodes	☑	☐	

6. Define trigger conditions for the Signal Tap nodes, as Defining Trigger Conditions on page 50 describes.

The number of channels available in the Signal Tap window waveform display is directly proportional to the number of logic elements (LEs) or adaptive logic modules (ALMs) in the device. Therefore, there is a physical restriction on the number of channels that are available for monitoring. Signals shown in blue text are post-fit node names. Signals shown in black text are pre-synthesis node names.

After successful Analysis and Elaboration, invalid signals appear in red. Unless you are certain that these signals are valid, remove them from the `.stp` file for correct operation. The Signal Tap Status Indicator also indicates if an invalid node name exists in the `.stp` file.

You can monitor signals only if a routing resource (row or column interconnects) exists to route the connection to the Signal Tap instance. For example, you cannot monitor signals that exist in the I/O element (IOE), because there are no direct routing resources from the signal in an IOE to a core logic element. For input pins, you can monitor the signal that is driving a logic array block (LAB) from an IOE, or, for output pins, you can monitor the signal from the LAB that is driving an IOE.

Note: The Intel 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`

2.4.5.2. Adding Simulator-Aware Signal Tap Nodes

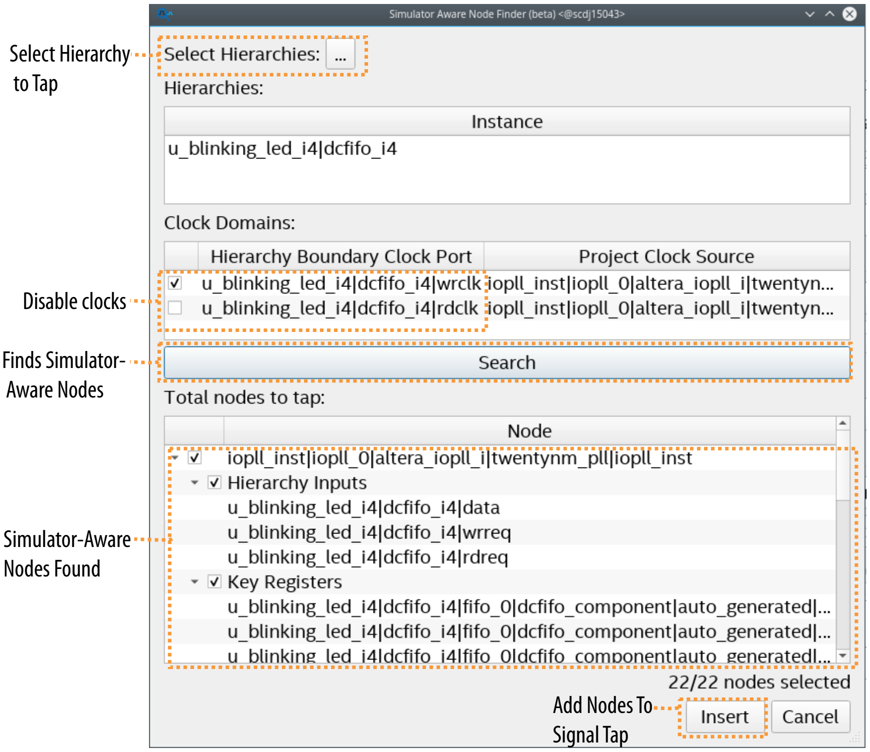
Note: This version of the Signal Tap simulator integration feature is a beta release. The following known limitations apply to this beta release:

- Supports only Verilog HDL simulation.
- Supports testbench generation only within the current project directory.

To automatically generate and add a list of simulator-aware signals to the Signal Tap **Node** list for Signal Tap and simulator monitoring, follow these steps:

1. To generate the pre-synthesis design netlist, click **Processing ➤ Start ➤ Start Analysis & Elaboration**.
2. In the Signal Tap logic analyzer, click **Edit ➤ Add Simulator Aware Nodes**. The **Simulator Aware Node Finder** opens, allowing you to specify the following options to find and add the minimum set of nodes to tap to for full visibility into the selected hierarchy's cone of logic:
    a. Click the **Select Hierarchies** button, select one or more design hierarchies that you want to tap, and then click **OK**. The clock domains in the hierarchy appear in the **Clock Domains** list.
    b. Under **Clock Domains**, enable only the domains of interest. If you select multiple clock domains, Signal Tap creates an instance for each domain.
    c. Click the **Search** button. All nodes required to provide full visibility into the selected hierarchy automatically appear enabled in the **Total nodes to tap** list. Disabling any of the simulator-aware nodes may reduce simulation visibility.
    d. Click the **Insert** button. The enabled signals in the **Total nodes to tap** list are copied to the Signal Tap **Node** list, and the acquisition clock updates according to the simulator-aware signal data. Refer to Add Simulator-Aware Node Finder Settings on page 49.

Figure 36. Simulator Aware Node Finder



Simulator Aware Node Finder (beta) <@scdj15043>

Select Hierarchies: ...

Hierarchies:

Instance
u_blinking_led_i4|dcfifo_i4

Clock Domains:

	Hierarchy Boundary Clock Port	Project Clock Source
☑	u_blinking_led_i4|dcfifo_i4|wrclk	iopll_inst|iopll_0|altera_iopll_i|twentyn...
□	u_blinking_led_i4|dcfifo_i4|rdclk	iopll_inst|iopll_0|altera_iopll_i|twentyn...

Search

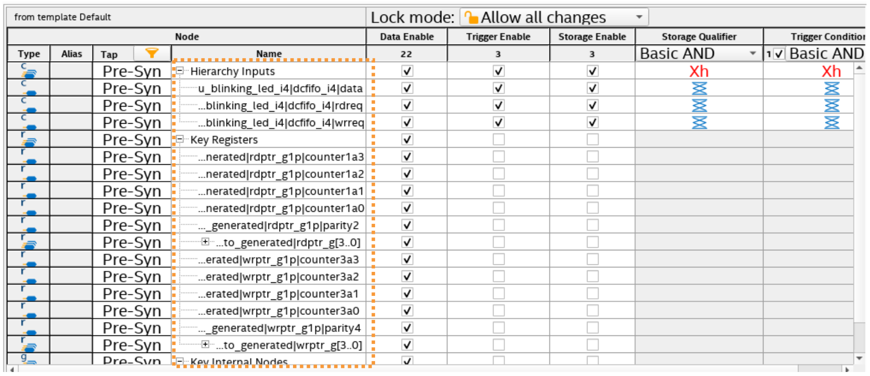
Total nodes to tap:

Node
☑ iopll_inst|iopll_0|altera_iopll_i|twentynm_pll|iopll_inst
☑ Hierarchy Inputs
u_blinking_led_i4|dcfifo_i4|data
u_blinking_led_i4|dcfifo_i4|wrreq
u_blinking_led_i4|dcfifo_i4|rdreq
☑ Key Registers
u_blinking_led_i4|dcfifo_i4|fifo_0|dcfifo_component|auto_generated|...
u_blinking_led_i4|dcfifo_i4|fifo_0|dcfifo_component|auto_generated|...
u_blinking_led_i4|dcfifo_i4|fifo_0|dcfifo_component|auto_generated|...

22/22 nodes selected

Insert Cancel

Figure 37. Simulator-Aware Nodes Copied to Signal Tap Window



from template Default

Lock mode: Allow all changes

Node				Data Enable	Trigger Enable	Storage Enable	Storage Qualifier	Trigger Conditio
Type	Alias	Tap	Name	22	3	3	Basic AND	1 ☑ Basic AND
c		Pre-Syn	⊟ Hierarchy Inputs	☑	☑	☑	Xh	Xh
c		Pre-Syn	u_blinking_led_i4|dcfifo_i4|data	☑	☑	☑	⧗	⧗
c		Pre-Syn	...blinking_led_i4|dcfifo_i4|rdreq	☑	☑	☑	⧗	⧗
c		Pre-Syn	...blinking_led_i4|dcfifo_i4|wrreq	☑	☑	☑	⧗	⧗
r		Pre-Syn	⊟ Key Registers	☑	□	□		
r		Pre-Syn	...nerated|rdptr_g1p|counter1a3	☑	□	□		
r		Pre-Syn	...nerated|rdptr_g1p|counter1a2	☑	□	□		
r		Pre-Syn	...nerated|rdptr_g1p|counter1a1	☑	□	□		
r		Pre-Syn	...nerated|rdptr_g1p|counter1a0	☑	□	□		
r		Pre-Syn	..._generated|rdptr_g1p|parity2	☑	□	□		
r		Pre-Syn	⊞ ...to_generated|rdptr_g[3..0]	☑	□	□		
r		Pre-Syn	...erated|wrptr_g1p|counter3a3	☑	□	□		
r		Pre-Syn	...erated|wrptr_g1p|counter3a2	☑	□	□		
r		Pre-Syn	...erated|wrptr_g1p|counter3a1	☑	□	□		
r		Pre-Syn	...erated|wrptr_g1p|counter3a0	☑	□	□		
r		Pre-Syn	..._generated|wrptr_g1p|parity4	☑	□	□		
r		Pre-Syn	⊞ ...to_generated|wrptr_g[3..0]	☑	□	□		
g		Pre-Syn	⊟ Key Internal Nodes	☑	□	□		

3. Modify trigger conditions for the Signal Tap nodes, as Defining Trigger Conditions on page 50 describes.

4. Compile the design and Signal Tap instance, Step 3: Compile the Design and Signal Tap Instances on page 80 describes.

5. Program the target hardware, as Step 4: Program the Target Hardware on page 83 describes.

6. Run the Signal Tap logic analyzer, as Step 5: Run the Signal Tap Logic Analyzer on page 83 describes.

7. Generate a simulation testbench from Signal Tap capture data, as Generating a Simulation Testbench from Signal Tap Data on page 96 describes.

2.4.5.2.1. Add Simulator-Aware Node Finder Settings

The following options are available for searching and adding simulator aware nodes to Signal Tap for the purpose of generating an RTL simulation testbench from Signal Tap data. The default values derive from Signal Tap signal data and are set correctly for most scenarios.

Table 10. Add Simulator Aware Node Finder Settings (Signal Tap Logic Analyzer)

Name	Description
Select Hierarchies	Specifies the design hierarchy from which to extract simulator-aware nodes. Select one or more design hierarchies that you want to tap. The clock domains of the hierarchy appear in the **Clock Domains** list. Only nodes from the hierarchy you specify are added.
Clock Domains	Specifies the clock domains to include in the simulator-aware node finder. Turn on only the domains that you want to include.
Search button	Starts the search for simulator-aware nodes according to the specifications in this dialog box. Search results appear in the **Total nodes to tap** list.
Total nodes to tap	Displays the results of the simulator-aware node name search, showing all of the names in the hierarchy enabled by default. Turn the node names on to include or off to exclude from the list of nodes added to Signal Tap. Disabling any of the simulator-aware nodes may reduce simulation visibility.
Insert Button	Copies the enabled signals in the **Total nodes to tap** list to the Signal Tap **Node** list, and the acquisition clock updates according to the simulator-aware signal data.

2.4.5.3. Adding Nios II Processor Signals with a Plug-In

You can use a plug-in to automatically add relevant signals for the Nios II processor for monitoring, rather than adding the signals manually with the Node Finder. The plug-in provides preset mnemonic tables for trigger creation and viewing, as well as the ability to disassemble code in captured data.

Note: This feature does not yet support the Nios V embedded processor.

The Nios II plug-in creates one mnemonic table in the **Setup** tab and two tables in the **Data** tab:

- **Nios II Instruction** (**Setup** tab)—capture all the required signals for triggering on a selected instruction address.
- **Nios II Instance Address** (**Data** tab)—display address of executed instructions in hexadecimal format or as a programming symbol name if defined in an optional Executable and Linking Format (`.elf`) file.
- **Nios II Disassembly** (**Data** tab)—display disassembled code from the corresponding address.

To add Nios II IP signals to the logic analyzer using a plug-in, perform the following steps after running Analysis and Elaboration on your design:

1. In the Signal Tap logic analyzer, right-click the node list, and then click **Add Nodes with Plug-In ➤ Nios II**.
2. Select the IP that contains the signals you want to monitor with the plug-in, and click **OK**.

- — If all the signals in the plug-in are available, a dialog box might appear, depending on the plug-in, where you can specify options for the plug-in.

3. With the Nios II plug-in, you can optionally select an `.elf` containing program symbols from your Nios II Integrated Development Environment (IDE) software design. Specify options for the selected plug-in, and click **OK**.

2.4.5.4. Signals Unavailable for Signal Tap Debugging

Some post-fit signals in your design are unavailable for Signal Tap debugging. The Node Finder's **Signal Tap: post-fitting** filter does not return nodes that are unavailable for Signal Tap debugging.

The following signal types are unavailable for Signal Tap debugging:

- **Post-fit output pins**—You cannot monitor a post-fit output or bidirectional pin directly. To make an output signal visible, monitor the register or buffer that drives the output pin.
- **Carry chain signals**—You cannot monitor 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 monitor the JTAG control (`TCK`, `TDI`, `TDO`, or `TMS`) signals.
- **LVDS**—You cannot monitor the data output from a serializer/deserializer (SERDES) block.
- **DQ**, **DQS signals**—You cannot directly monitor the `DQ` or `DQS` signals in a DDR or DDRII design.

2.4.6. Defining Trigger Conditions

By default, the Signal Tap logic analyzer captures data continuously from the signals you specify while the logic analyzer is running. To capture and store only specific signal data, you can specify conditions that *trigger* the start or stop of data capture. A trigger activates—that is, the logic analyzer stops and displays the data—when the signals you specify reach the trigger conditions that you define.

The Signal Tap logic analyzer allows you to define trigger conditions that range from very simple, such as the rising edge of a single signal, to very complex, involving groups of signals, extra logic, and multiple conditions. Additionally, you can specify Power-Up Triggers to capture data from trigger events occurring immediately after the device enters user-mode after configuration.

2.4.6.1. Basic Trigger Conditions

If you select the **Basic AND** or **Basic OR** trigger type, you must specify the trigger pattern for each signal that you add.

To specify the trigger pattern, right-click the **Trigger Conditions** column and click **Don’t Care**, **Low**, **High**, **Falling Edge** , **Rising Edge**, or **Either Edge**.

For buses, type a pattern in binary, or right-click and select **Insert Value** to enter the pattern in other number formats. Note that you can enter X to specify a set of “don’t care” values in either your hexadecimal or your binary string. For signals in the `.stp` file that have an associated mnemonic table, you can right-click and select an entry from the table to specify pre-defined conditions for the trigger.

When you add signals through plug-ins, you can create basic triggers using predefined mnemonic table entries. For example, with the Nios II plug-in, if you specify an `.elf` file from your Nios II IDE design, you can type the name of a function from your Nios II code. The logic analyzer triggers when the Nios II instruction address matches the address of the code function name that you specify.

Data capture stops and the logic analyzer stores the data in the buffer when the logical AND of all the signals for a given trigger condition evaluates to TRUE.

2.4.6.2. Nested Trigger Conditions

When you specify a set of signals as a nested group (group of groups) with the **Basic OR** trigger type, the logic analyzer generates an advanced trigger condition. This condition sorts signals within groups to minimize the need to recompile your design. If you always retain the parent-child relationship of nodes, the advanced trigger condition does not change. You can modify the sibling relationships of nodes, without requiring recompilation.

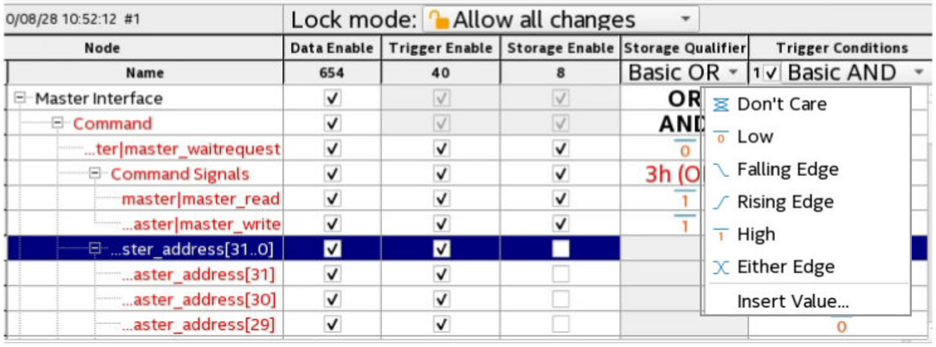
The evaluation precedence of a nested trigger condition starts at the bottom-level with the leaf-groups. The logic analyzer uses the resulting logic value to compute the parent group's logic value. If you manually set the value of a group, the logic value of the group's members doesn't influence the result of the group trigger.

To create a nested trigger condition:

1. Select **Basic OR** under **Trigger Conditions**.
2. In the **Setup** tab, select several nodes. Include groups in your selection.
3. Right-click the **Setup** tab and select **Group**.
4. Select the nested group and right-click to set a group trigger condition that applies the reduction **AND**, **OR**, **NAND**, **NOR**, **XOR**, **XNOR**, or logical **TRUE** or **FALSE**.

   Note: You can only select OR and AND group trigger conditions for bottom-level groups (groups with no groups as children).

Figure 38. Applying Trigger Condition to Nested Group



0/08/28 10:52:12 #1 — Lock mode: Allow all changes

Node	Data Enable	Trigger Enable	Storage Enable	Storage Qualifier	Trigger Conditions
Name	654	40	8	Basic OR	1 ✓ Basic AND
⊟ Master Interface	✓	✓	✓	OR	
⊟ Command	✓	✓	✓	AND	
...ter|master_waitrequest	✓	✓	✓	0	
⊟ Command Signals	✓	✓	✓	3h (O	
master|master_read	✓	✓	✓	1	
...aster|master_write	✓	✓	✓	1	
⊟ ...ster_address[31..0]	✓	✓	□		
...aster_address[31]	✓	✓	□		
...aster_address[30]	✓	✓	□		
...aster_address[29]	✓	✓	□		0

2.4.6.3. Comparison Trigger Conditions

The **Comparison** trigger allows you to compare multiple grouped bits of a bus to an expected integer value by specifying simple comparison conditions on the bus node. The **Comparison** trigger preserves all the trigger conditions that the **Basic OR** trigger includes. You can use the **Comparison** trigger in combination with other triggers. You can also switch between **Basic OR** trigger and **Comparison** trigger at run-time, without the need for recompilation.

Signal Tap logic analyzer supports the following types of **Comparison** trigger conditions:

- **Single-value comparison**—compares a bus node’s value to a numeric value that you specify. Use one of these operands for comparison: >, >=, ==, <=, <. Returns 1 when the bus node matches the specified numeric value.
- **Interval check**—verifies whether a bus node’s value confines to an interval that you define. Returns 1 when the bus node's value lies within the specified bounded interval.

Follow these rules when using the **Comparison** trigger condition:

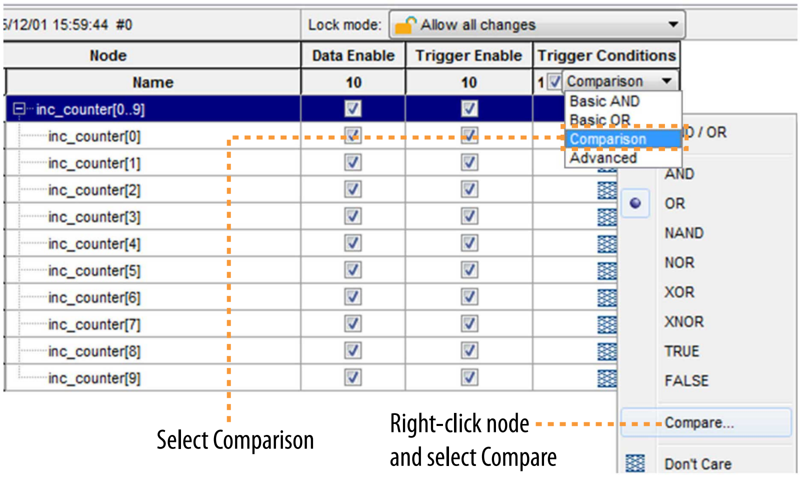
- Apply the **Comparison** trigger only to bus nodes consisting of leaf nodes.
- Do not form sub-groups within a bus node.
- Do not enable or disable individual trigger nodes within a bus node.
- Do not specify comparison values (in case of single-value comparison) or boundary values (in case of interval check) exceeding the selected node’s bus-width.

2.4.6.3.1. Specifying the Comparison Trigger Conditions

Follow these steps to specify the **Comparison** trigger conditions:

1. From the **Setup** tab, select **Comparison** under **Trigger Conditions**.
2. Right-click the node in the trigger editor, and select **Compare**.
3. Select the **Comparison type** from the Compare window.
   - — If you choose **Single-value comparison** as your comparison type, specify the operand and value.
   - — If you choose **Interval check** as your comparison type, provide the lower and upper bound values for the interval.

Figure 39. Selecting the Comparison Trigger Condition



5/12/01 15:59:44 #0

Lock mode: Allow all changes

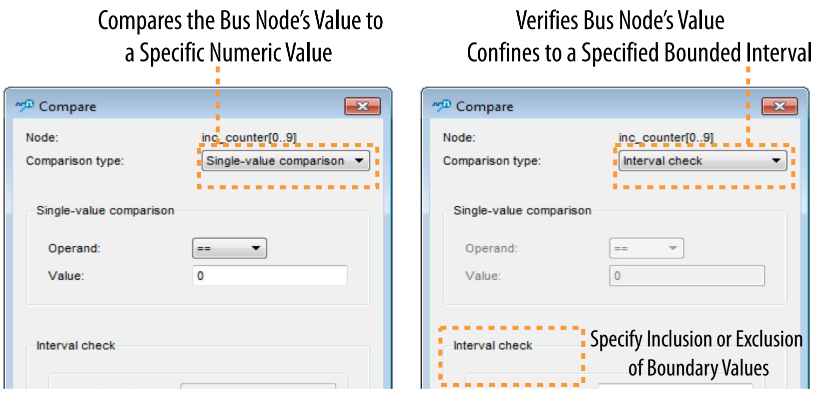
Node	Data Enable	Trigger Enable	Trigger Conditions
Name	10	10	1 ☑ Comparison
⊟ inc_counter[0..9]	☑	☑	
inc_counter[0]	☑	☑	
inc_counter[1]	☑	☑	
inc_counter[2]	☑	☑	▨
inc_counter[3]	☑	☑	▨
inc_counter[4]	☑	☑	▨
inc_counter[5]	☑	☑	▨
inc_counter[6]	☑	☑	▨
inc_counter[7]	☑	☑	▨
inc_counter[8]	☑	☑	▨
inc_counter[9]	☑	☑	▨

Select Comparison

Right-click node and select Compare

You can also specify if you want to include or exclude the boundary values.

Figure 40. Specifying the Comparison Values



Compares the Bus Node's Value to a Specific Numeric Value

Verifies Bus Node's Value Confines to a Specified Bounded Interval

Compare

Node: inc_counter[0..9]
Comparison type: Single-value comparison

Single-value comparison
Operand: ==
Value: 0

Interval check

Compare

Node: inc_counter[0..9]
Comparison type: Interval check

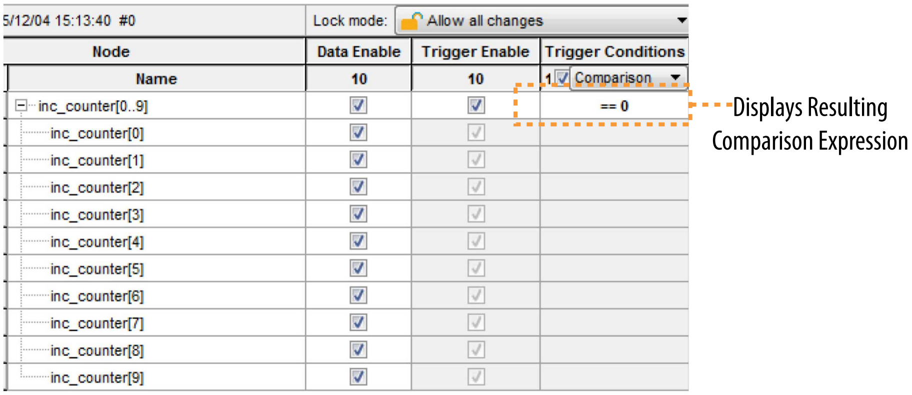
Single-value comparison
Operand: ==
Value: 0

Interval check

Specify Inclusion or Exclusion of Boundary Values

4. Click **OK**. The trigger editor displays the resulting comparison expression in the group node condition text box.

Figure 41. Resulting Comparison Condition in Text Box



5/12/04 15:13:40 #0 Lock mode: Allow all changes

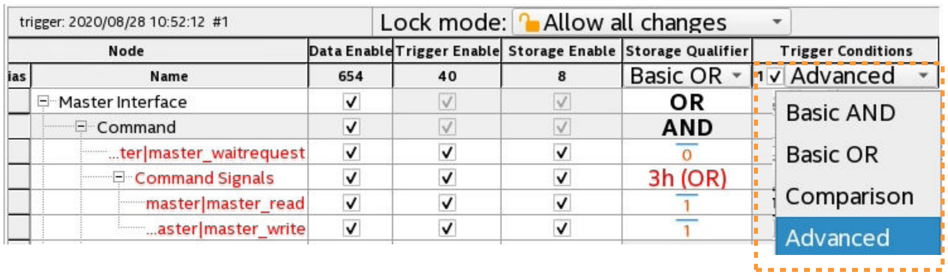
Node Name	Data Enable 10	Trigger Enable 10	Trigger Conditions 1 ☑ Comparison
⊟ inc_counter[0..9]	☑	☑	== 0
inc_counter[0]	☑	☑	
inc_counter[1]	☑	☑	
inc_counter[2]	☑	☑	
inc_counter[3]	☑	☑	
inc_counter[4]	☑	☑	
inc_counter[5]	☑	☑	
inc_counter[6]	☑	☑	
inc_counter[7]	☑	☑	
inc_counter[8]	☑	☑	
inc_counter[9]	☑	☑	

Displays Resulting Comparison Expression

2.4.6.4. Advanced Trigger Conditions

To capture data for a given combination of conditions, build an advanced trigger. The Signal Tap logic analyzer provides the **Advanced Trigger** tab, which helps you build a complex trigger expression using a GUI. Open the **Advanced Trigger** tab by selecting **Advanced** in the **Trigger Conditions** list.

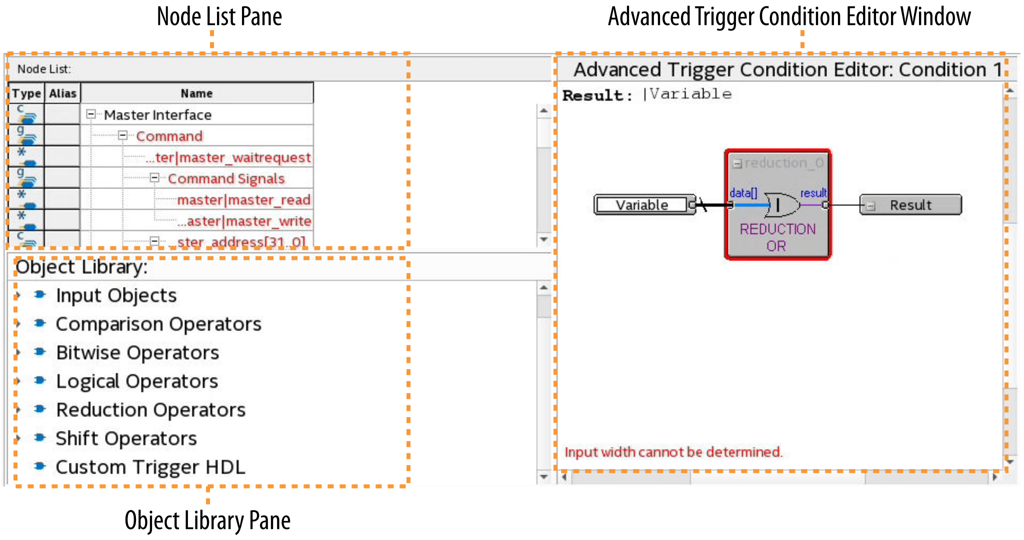
Figure 42. Accessing the Advanced Trigger Condition Tab



trigger: 2020/08/28 10:52:12 #1 Lock mode: Allow all changes

Node ias	 Name	Data Enable 654	Trigger Enable 40	Storage Enable 8	Storage Qualifier Basic OR	Trigger Conditions 1 ☑ Advanced
	⊟ Master Interface	☑	☑	☑	OR	
	⊟ Command	☑	☑	☑	AND	
	...ter|master_waitrequest	☑	☑	☑	0	
	⊟ Command Signals	☑	☑	☑	3h (OR)	
	master|master_read	☑	☑	☑	1	
	...aster|master_write	☑	☑	☑	1	

Figure 43. Advanced Trigger Condition Tab



Input width cannot be determined.

To build a complex trigger condition in an expression tree, drag-and-drop operators from the **Object Library** pane and the **Node List** pane into the **Advanced Trigger Configuration Editor** window.

To configure the operators' settings, double-click or right-click the operators that you placed and click **Properties**.

Table 11. Advanced Triggering Operators

Category	Name
Signal Detection	Edge and Level Detector
Input Objects	Bit Bit Value Bus Bus Value
Comparison	Less Than Less Than or Equal To Equality Inequality Greater Than or Equal To Greater Than
Bitwise	Bitwise Complement Bitwise AND Bitwise OR Bitwise XOR
Logical	Logical NOT Logical AND Logical OR Logical XOR
Reduction	Reduction AND Reduction OR

continued...

Category	Name
	Reduction XOR
Shift	Left Shift Right Shift
Custom Trigger HDL	

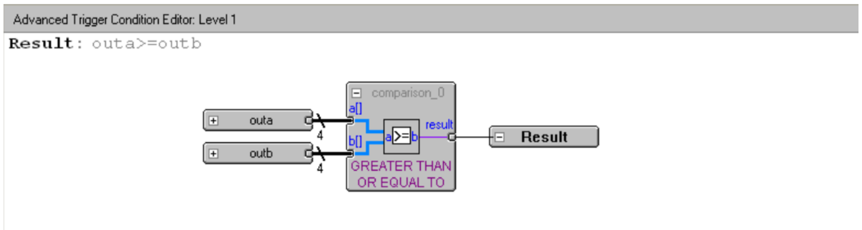
Adding many objects to the Advanced Trigger Condition Editor can make the work space cluttered and difficult to read. To keep objects organized while you build your advanced trigger condition, use the shortcut menu and select **Arrange All Objects**. Alternatively, use the **Zoom-Out** command to fit more objects into the **Advanced Trigger Condition Editor** window.

2.4.6.4.1. Examples of Advanced Triggering Expressions

The following examples show how to use advanced triggering:

Figure 44. Bus outa Is Greater Than or Equal to Bus outb

Trigger when bus `outa` is greater than or equal to `outb`.

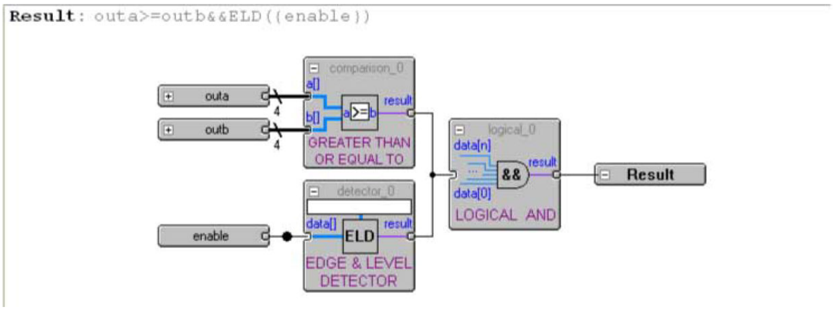


Advanced Trigger Condition Editor: Level 1

```
Result: outa>=outb
```

Figure 45. Enable Signal Has a Rising Edge

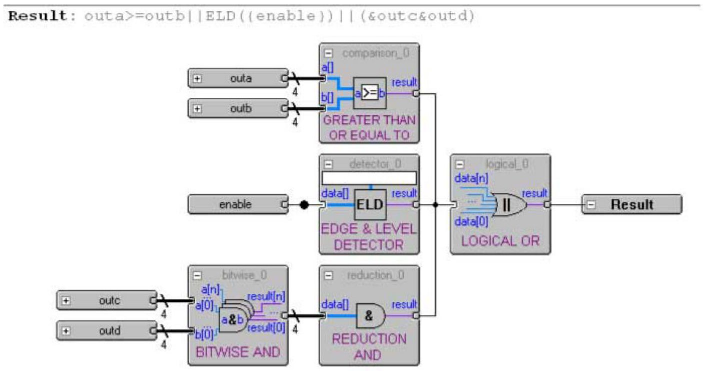
Trigger when bus `outa` is greater than or equal to bus `outb`, and when the enable signal has a rising edge.



```
Result: outa>=outb&&ELD((enable))
```

Figure 46. 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.



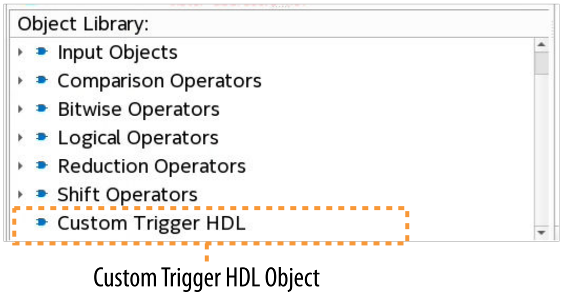
Result: outa>=outb||ELD((enable))||(&outc&outd)

2.4.6.5. Custom Trigger HDL Object

The Signal Tap logic analyzer supports use of your own HDL module to define a custom trigger condition. You can use the Custom Trigger HDL object to simulate your triggering logic and ensure that the logic itself is not faulty. Additionally, you can monitor instances of modules anywhere in the hierarchy of your design, without having to manually route all the necessary connections.

The **Custom Trigger HDL** object appears in the **Object Library** pane of the **Advanced Trigger** editor.

Figure 47. Object Library

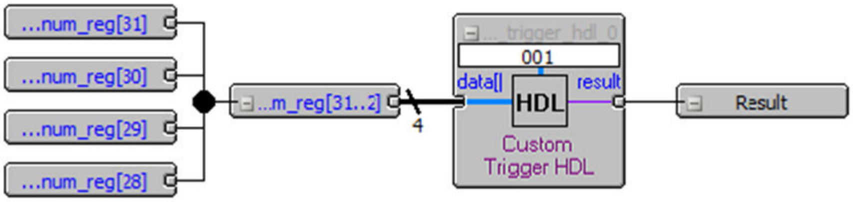


2.4.6.5.1. Using the Custom Trigger HDL Object

To define a custom trigger flow:

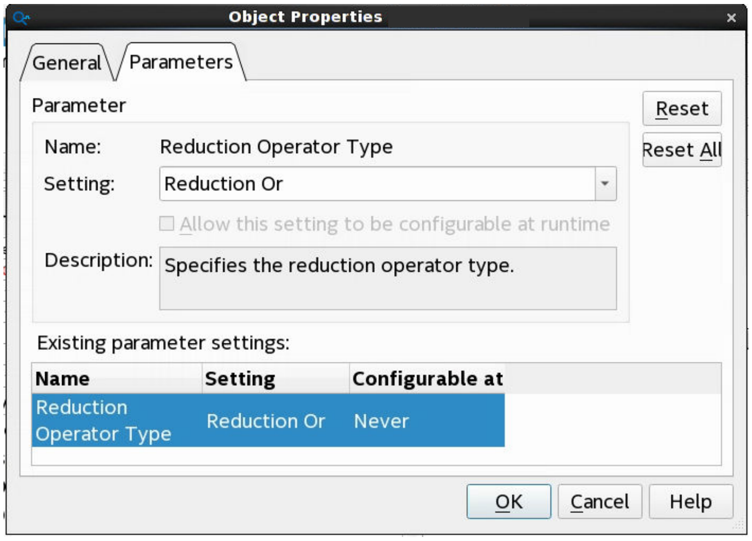
1. Select the trigger you want to edit.
2. Open the **Advanced Trigger** tab by selecting **Advanced** in the **Trigger Conditions** list.
3. Add to your project the Verilog HDL or VHDL source file that contains the trigger module using the **Project Navigator**.
4. Implement the inputs and outputs that your Custom Trigger HDL module requires.
5. Drag in your **Custom Trigger HDL** object and connect the object's data input bus and result output bit to the final trigger result.

Figure 48. Custom Trigger HDL Object



6. Right-click your **Custom Trigger HDL** object and configure the object's properties.

Figure 49. Configure Object Properties



Parameter

Name: Reduction Operator Type

Setting: Reduction Or

□ Allow this setting to be configurable at runtime

Description: Specifies the reduction operator type.

Existing parameter settings:

Name	Setting	Configurable at
Reduction Operator Type	Reduction Or	Never

7. Compile your design.
8. Acquire data with Signal Tap using your custom Trigger HDL object.

Example 1. Verilog HDL Triggers

The following trigger uses configuration bitstream:

```
module test_trigger
    (
        input acq_clk, reset,
        input[3:0] data_in,
        input[1:0] pattern_in,
        output reg trigger_out
    );
    always @(pattern_in) begin
        case (pattern_in)
            2'b00:
                trigger_out = &data_in;
            2'b01:
                trigger_out = |data_in;
            2'b10:
                trigger_out = 1'b0;
            2'b11:
                trigger_out = 1'b1;
        endcase
    end
endmodule
```

This trigger does not have configuration bitstream:

```
module test_trigger_no_bs
    (
        input acq_clk, reset,
        input[3:0] data_in,
        output reg trigger_out
    );
    assign trigger_out = &data_in;
endmodule
```

2.4.6.5.2. Required Inputs and Outputs of Custom Trigger HDL Module

Table 12. Custom Trigger HDL Module Required Inputs and Outputs

Name	Description	Input/Output	Required/ Optional
`acq_clk`	Acquisition clock that Signal Tap uses	Input	Required
`reset`	Reset that Signal Tap uses when restarting a capture.	Input	Required
`data_in`	• Data input you connect in the Advanced Trigger editor. • Data your module uses to trigger.	Input	Required
`pattern_in`	• Module's input for the configuration bitstream property. • Runtime configurable property that you can set from Signal Tap GUI to change the behavior of your trigger logic.	Input	Optional
`trigger_out`	Output signal of your module that asserts when trigger conditions met.	Output	Required

2.4.6.5.3. Custom Trigger HDL Module Properties

Table 13. Custom Trigger HDL Module Properties

Property	Description
Custom HDL Module Name	Module name of the triggering logic.
Configuration Bitstream	• Allows to create trigger logic that you can configure at runtime, based upon the value of the configuration bitstream. • The Signal Tap logic analyzer reads the configuration bitstream property as binary, therefore the bitstream must contain only the characters 1 and `0`. • The bit-width (number of 1s and 0s) must match the `pattern_in` bit width. • A blank configuration bitstream implies that the module does not have a `pattern_in` input.
Pipeline	Specifies the number of pipeline stages in the triggering logic. For example, if after receiving a triggering input the LA needs three clock cycles to assert the trigger output, you can denote a pipeline value of three.

2.4.6.6. Specify Trigger Position

You can specify the amount of data the logic analyzer acquires before and after a trigger event. Positions for Runtime and Power-Up triggers are separate.

The Signal Tap logic analyzer offers three pre-defined ratios of pre-trigger data to post-trigger data:

- **Pre**—saves signal activity that occurred after the trigger (12% pre-trigger, 88% post-trigger).
- **Center**—saves 50% pre-trigger and 50% post-trigger data.
- **Post**—saves signal activity that occurred before the trigger (88% pre-trigger, 12% post-trigger).

These pre-defined ratios apply to both non-segmented buffers and each segment of a buffer.

2.4.6.6.1. Post-fill Count

In a custom state-based triggering flow with the `segment_trigger` and `trigger` buffer control actions, you can use the `post-fill_count` argument to specify a custom trigger position.

- If you do not use the `post-fill_count` argument, the trigger position for the affected buffer defaults to the trigger position you specified in the **Setup** tab.
- In the `trigger` buffer control action (for non-segmented buffers), `post-fill_count` specifies the number of samples to capture before stopping data acquisition.
- In the `segment_trigger` buffer control action (for segmented buffer), `post-fill_count` specifies a data segment.

Note: In the case of `segment_trigger`, acquisition of the current buffer stops immediately if a subsequent triggering action is issued in the next state, regardless of the current buffer's post-fill count. The logic analyzer discards the remaining unfilled post-count acquisitions in the current buffer, and displays them as grayed-out samples in the data window.

When the Signal Tap data window displays the captured data, the trigger position appears as the number of post-count samples from the end of the acquisition segment or buffer.

Sample Number of Trigger Position = (*N* – *Post-Fill Count*)

In this case, *N* is the sample depth of either the acquisition segment or non-segmented buffer.

Related Information

Buffer Control Actions on page 70

2.4.6.7. Power-Up Triggers

Power-Up Triggers capture events that occur during device initialization, immediately after you power or reset the FPGA.

The typical use of Signal Tap logic analyzer is triggering events that occur during normal device operation. You start an analysis manually once the target device fully powers on and the JTAG connection for the device is available. With Signal Tap Power-Up Trigger feature, the Signal Tap logic analyzer captures data immediately after device initialization.

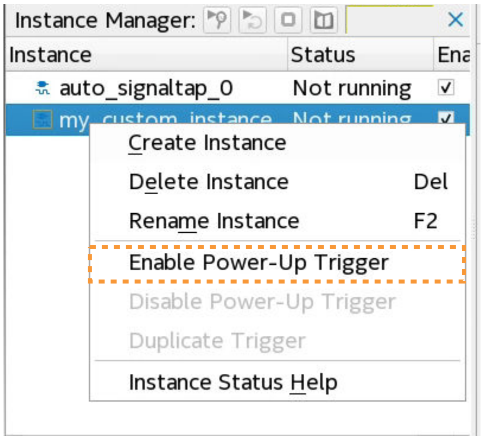
You can add a different Power-Up Trigger to each logic analyzer instance in the **Signal Tap Instance Manager** pane.

2.4.6.7.1. Enabling a Power-Up Trigger

To enable the Power-Up Trigger for Signal Tap instance:

- In the Instance Manager, right-click the Signal Tap instance and click **Enable Power-Up Trigger**.

Figure 50. Enabling Power-Up Trigger in Signal Tap Instance Manager



Power-Up Trigger appears as a child instance below the name of the selected instance. The node list displays the default trigger conditions.

To disable a Power-Up Trigger, right-click the instance and click **Disable Power-Up Trigger**.

2.4.6.7.2. Configuring Power-Up Trigger Conditions

- Any change that you make to a Power-Up Trigger conditions requires that you recompile the Signal Tap logic analyzer instance, even if a similar change to the Runtime Trigger conditions does not require a recompilation.
- You can also force trigger conditions with the In-System Sources and Probes in conjunction with the Signal Tap logic analyzer. 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 on page 144

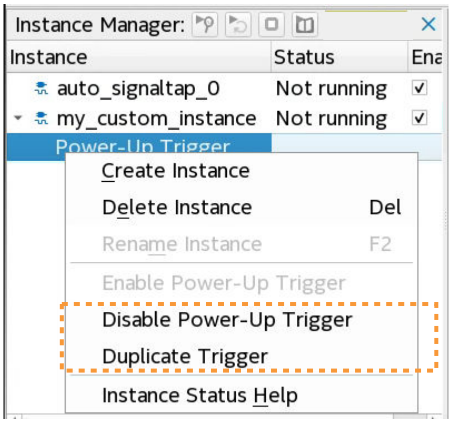
2.4.6.7.3. Managing Signal Tap Instances with Run-Time and Power-Up Trigger Conditions

On instances that have two types of trigger conditions, Power-Up Trigger conditions are color coded light blue, while Run-Time Trigger conditions remain white.

To switch between the trigger conditions of the Power-Up Trigger and the Run-Time Trigger, double-click the instance name or the Power-Up Trigger name in the **Instance Manager**.

To copy trigger conditions from a Run-Time Trigger to a Power-Up Trigger or vice versa, right-click the trigger name in the **Instance Manager** and click **Duplicate Trigger**. Alternatively, select the trigger name and click **Edit ➤ Duplicate Trigger**.

Figure 51. Instance Manager Commands



Note: Run-time trigger conditions allow fewer adjustments than power-up trigger conditions.

2.4.6.8. External Triggers

External trigger inputs allow you to trigger the Signal Tap logic analyzer from an external source.

The external trigger input behaves like trigger condition 0, in that the condition must evaluate to TRUE before the logic analyzer evaluates any other trigger conditions.

The Signal Tap logic analyzer supplies a signal to trigger external devices or other logic analyzer instances. These features allow you to synchronize external logic analysis equipment with the internal logic analyzer. Power-Up Triggers can use the external triggers feature, but they must use the same source or target signal as their associated Run-Time Trigger.

You can use external triggers to perform cross-triggering on a hard processor system (HPS):

- The processor debugger allows you 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.
- The processor debugger in combination with the Signal Tap external trigger feature allow you to develop a dynamic combination of cross-trigger behaviors.
- You can implement a system-level debugging solution for an Intel FPGA SoC by using the cross-triggering feature with the ARM Development Studio 5 (DS-5) software.

2.4.6.9. Trigger Condition Flow Control

The Trigger Condition Flow Control allows you to define the relationship between a set of triggering conditions. Signal Tap logic analyzer **Signal Configuration** pane offers two flow control mechanisms for organizing trigger conditions:

- **Sequential Triggering**—default triggering flow. Sequential triggering allows you to define up to 10 triggering levels that must be satisfied before the acquisition buffer finishes capturing.
- **State-Based Triggering**—gives the greatest control over your acquisition buffer. Custom-based triggering allows you to organize trigger conditions into states based on a conditional flow that you define.

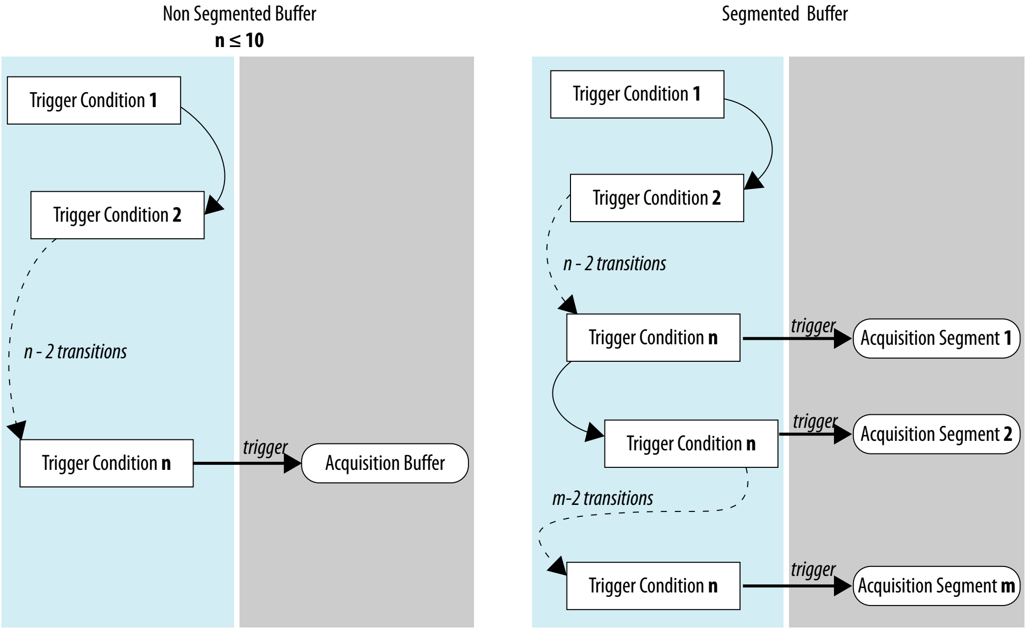
You can use sequential or state based triggering with either a segmented or a non-segmented buffer.

2.4.6.10. Sequential Triggering

When you specify a sequential trigger the Signal Tap logic analyzer sequentially evaluates each the conditions. The sequential triggering flow allows you to cascade up to 10 levels of triggering conditions.

When the last triggering condition evaluates to TRUE, the Signal Tap logic analyzer starts the data acquisition. For segmented buffers, every acquisition segment after the first starts on the last condition that you specified. The Signal Tap **Node** annotates this final condition column with Seg if a segmented buffer is enabled. The Simple Sequential Triggering feature allows you to specify basic triggers, comparison triggers, advanced triggers, or a mix of all three. The following figure illustrates the simple sequential triggering flow for non-segmented and segmented buffers. The acquisition buffer starts capture when all n triggering levels are satisfied, where $n \leq 10$.

Figure 52. Sequential Triggering Flow



The Signal Tap logic analyzer considers external triggers as level `0`, evaluating external triggers before any other trigger condition.

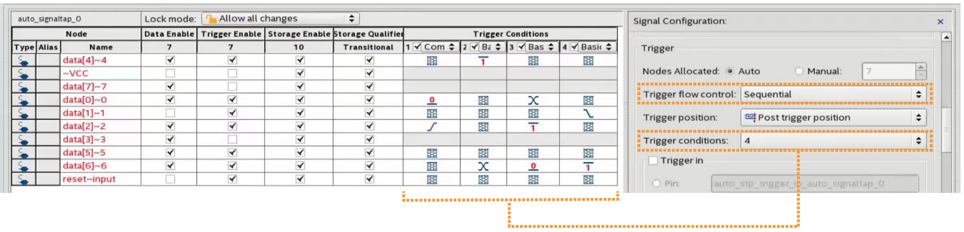
2.4.6.10.1. Configuring the Sequential Triggering Flow

To configure Signal Tap logic analyzer for sequential triggering:

1. On **Trigger Flow Control**, select **Sequential**
2. On **Trigger Conditions**, select the number of trigger conditions from the drop-down list.
   The **Node List** pane now displays the same number of trigger condition columns.
3. Configure each trigger condition in the **Node List** pane.

   You can enable/disable any trigger condition from the column header.

Figure 53. Sequential Triggering Flow Configuration



auto_signaltap_0 Lock mode: Allow all changes

Node			Data Enable	Trigger Enable	Storage Enable	Storage Qualifier	Trigger Conditions			
Type	Alias	Name	7	7	10	Transitional	1 ☑ Com	2 ☑ Ba	3 ☑ Bas	4 ☑ Basic
		data[4]~4	☑	☑	☑	☑	▨	1	▨	▨
		~VCC	☐	☐	☑	☑				
		data[7]~7	☑	☐	☑	☑				
		data[0]~0	☑	☑	☑	☑	0	▨	X	▨
		data[1]~1	☐	☑	☑	☑	▨	▨	▨	\
		data[2]~2	☑	☑	☑	☑	/	▨	1	▨
		data[3]~3	☑	☐	☑	☑				
		data[5]~5	☑	☑	☑	☑	▨	▨	▨	▨
		data[6]~6	☑	☑	☑	☑	▨	X	0	1
		reset~input	☐	☑	☑	☑	▨	▨	▨	▨

Signal Configuration:

Trigger

Nodes Allocated: ● Auto ○ Manual: 7

Trigger flow control: Sequential

Trigger position: Post trigger position

Trigger conditions: 4

☐ Trigger in

○ Pin: auto_stp_trigger_in_auto_signaltap_0

2.4.6.11. State-Based Triggering

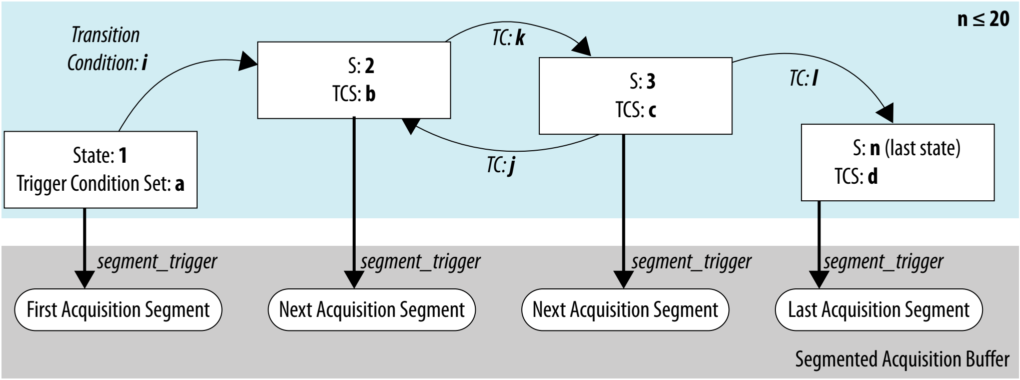
With state-based triggering, a state diagram organizes the events that trigger the acquisition buffer. The states capture all actions that the acquisition buffer performs, and each state contains conditional expressions that define transition conditions.

Custom state-based triggering grants control over triggering condition arrangement. Because the logic analyzer only captures samples of interest, custom state-based triggering allows for more efficient use of the space available in the acquisition buffer.

To help you describe the relationship between triggering conditions, the state-based triggering flow provides tooltips in the GUI. Additionally, you can use the Signal Tap Trigger Flow Description Language, which is based upon conditional expressions.

Each state allows you to define a set of conditional expressions. Conditional expressions are Boolean expressions that depend on a combination of triggering conditions, counters, and status flags. You configure the triggering conditions within the **Setup** tab. The Signal Tap logic analyzer custom-based triggering flow provides counters and status flags.

Figure 54. State-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 an optional count that specifies the number of samples the buffer captures before the logic analyzer stops acquisition of the current segment. The count argument allows you to control the amount of data the buffer captures before and after a triggering event occurs.

Resource manipulation actions allow you to increment and decrement counters or set and clear status flags. The logic analyzer uses counter and status flag resources as optional inputs in conditional expressions. Counters and status flags are useful for counting the number of occurrences of certain events and for aiding in triggering flow control.

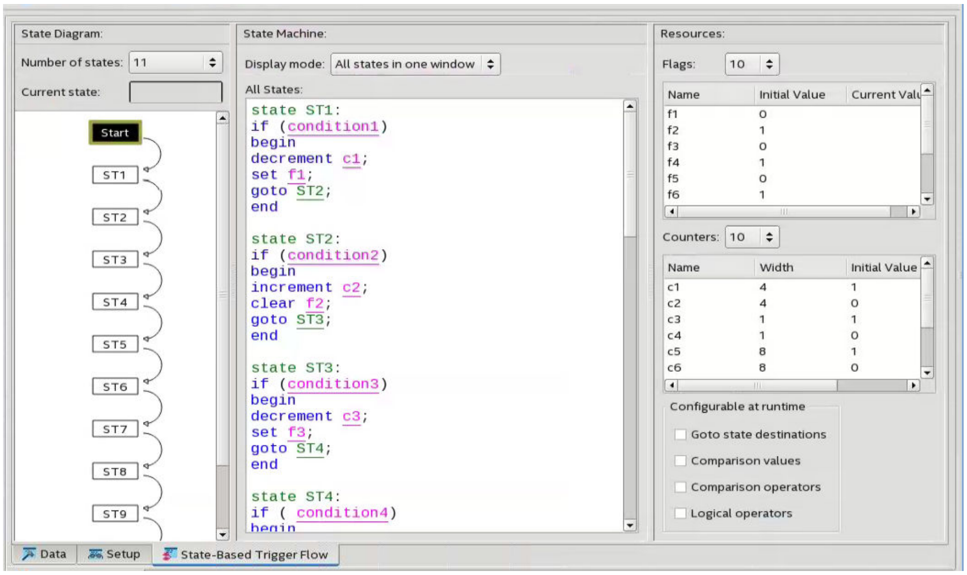
The state-based triggering flow allows you to capture a sequence of events that may not necessarily be contiguous in time. For example, a communication transaction between two devices that includes a hand shaking protocol containing a sequence of acknowledgments.

2.4.6.11.1. State-Based Triggering Flow Tab

The **State-Based Trigger Flow** tab is the control interface for the custom state-based triggering flow.

This tab is only available when you select **State-Based** on the **Trigger Flow Control** list. If you specify **Trigger Flow Control** as **Sequential**, the **State-Based Trigger Flow** tab is not visible.

Figure 55. State-Based Triggering Flow Tab



State Diagram:

Number of states: 11

Current state:

State Machine:

Display mode: All states in one window

All States:

```
state ST1:
if (condition1)
begin
decrement c1;
set f1;
goto ST2;
end

state ST2:
if (condition2)
begin
increment c2;
clear f2;
goto ST3;
end

state ST3:
if (condition3)
begin
decrement c3;
set f3;
goto ST4;
end

state ST4:
if ( condition4)
begin
```

Resources:

Flags: 10

Name	Initial Value	Current Valu
f1	0	
f2	1	
f3	0	
f4	1	
f5	0	
f6	1	

Counters: 10

Name	Width	Initial Value
c1	4	1
c2	4	0
c3	1	1
c4	1	0
c5	8	1
c6	8	0

Configurable at runtime

- □ Goto state destinations
- □ Comparison values
- □ Comparison operators
- □ Logical operators

The **State-Based Trigger Flow** tab contains three panes:

2.4.6.11.2. State Machine Pane

The **State Machine** pane contains the text entry boxes where you define the triggering flow and actions associated with each state.

- You can define the triggering flow using the Signal Tap Trigger Flow Description Language, a simple language based on "if-else" conditional statements.
- Tooltips appear when you move the mouse over the cursor, to guide command entry into the state boxes.
- The GUI provides a syntax check on your flow description in real-time and highlights any errors in the text flow.

The State Machine description text boxes default to show one text box per state. You can also have the entire flow description shown in a single text field. This option can be useful when copying and pasting a flow description from a template or an external text editor. To toggle between one window per state, or all states in one window, select the appropriate option under **State Display mode.**

Related Information

Signal Tap Trigger Flow Description Language on page 66

2.4.6.11.3. Resources Pane

The **Resources** pane allows you to declare status flags and counters for your Custom Triggering Flow's conditional expressions.

- You can increment/decrement counters or set/clear status flags within your triggering flow.
- You can specify up to 20 counters and 20 status flags.
- To initialize counter and status flags, right-click the row in the table and select **Set Initial Value.**
- To specify a counter width, right-click the counter in the table and select **Set Width**.
- To assist in debugging your trigger flow specification, the logic analyzer dynamically updates counters and flag values after acquisition starts.

The **Configurable at runtime** settings allow you to control which options can change at runtime without requiring a recompilation.

Table 14. Runtime Reconfigurable Settings, State-Based Triggering Flow

Setting	Description
Destination of `goto` action	Allows you to modify the destination of the state transition at runtime.
Comparison values	Allows you to modify comparison values in Boolean expressions at runtime. In addition, you can modify the `segment_trigger` and trigger action post-fill count argument at runtime.
Comparison operators	Allows you to modify the operators in Boolean expressions at runtime.
Logical operators	Allows you to modify the logical operators in Boolean expressions at runtime.

Related Information

- Performance and Resource Considerations on page 82
- Runtime Reconfigurable Options on page 87

2.4.6.11.4. State Diagram Pane

The **State Diagram** pane provides a graphical overview of your triggering flow. this pane displays the number of available states and the state transitions. To adjust the number of available states, use the menu above the graphical overview.

2.4.6.11.5. Signal Tap Trigger Flow Description Language

The Trigger Flow Description Language is based on a list of conditional expressions per state to define a set of actions.

To describe the actions that the logic analyzer evaluates when a state is reached, follow this syntax:

Syntax of Trigger Flow Description Language

```
state <state_label>:
    <action_list>
    if (<boolean_expression>)
        <action_list>
    [else if (<boolean_expression>)
        <action_list>]
    [else
        <action_list>]
```

- Non-terminals are delimited by "<>".
- Optional arguments are delimited by "[]".
- The priority for evaluation of conditional statements is from top to bottom.
- The Trigger Flow Description Language allows multiple `else if` conditions.

<state_label> on page 66

<boolean_expression> on page 67

<action_list> on page 68

Trigger that Skips Clock Cycles after Hitting Condition on page 68

Storage Qualification with Post-Fill Count Value Less than m on page 69

Resource Manipulation Action on page 70

Buffer Control Actions on page 70

State Transition Action on page 70

Related Information

Custom State-Based Triggering Flow Examples on page 112

<state_label>

Identifies a given state. You use the state label to start describing the actions the logic analyzer evaluates once said state is reached. You can also use the state label with the `goto` command.

The state description header syntax is:
`state` *<state_label>*

The description of a state ends with the beginning of another state or the end of the whole trigger flow description.

<boolean_expression>

Collection of operators and operands that evaluate into a Boolean result. The operators can be logical or relational. Depending on the operator, the operand can reference a trigger condition, a counter and a register, or a numeric value. To group a set of operands within an expression, you use parentheses.

Table 15. Logical Operators

Logical operators accept any boolean expression as an operand.

Operator	Description	Syntax
!	NOT operator	`! expr1`
&&	AND operator	`expr1 && expr2`
||	OR operator	`expr1 || expr2`

Table 16. Relational Operators

You use relational operators on counters or status flags.

Operator	Description	Syntax
>	Greater than	`<identifier> > <numerical_value>`
>=	Greater than or Equal to	`<identifier> >= <numerical_value>`
==	Equals	`<identifier> == <numerical_value>`
!=	Does not equal	`<identifier> != <numerical_value>`
<=	Less than or equal to	`<identifier> <= <numerical_value>`
<	Less than	`<identifier> < <numerical_value>`

Notes to table:
1. *<identifier>* indicates a counter or status flag.
2. *<numerical_value>* indicates an integer.

Note:

- The *<boolean_expression>* in an `if` statement can contain a single event or multiple event conditions.
- When the boolean expression evaluates TRUE, the logic analyzer evaluates all the commands in the *<action_list>* concurrently.

<action_list>

List of actions that the logic analyzer performs within a state once a condition is satisfied.

- Each action must end with a semicolon (`;`).
- If you specify more than one action within an `if` or an `else if` clause, you must delimit the `action_list` with `begin` and `end` tokens.

Possible actions include:

Buffer Control Actions

Actions that control the acquisition buffer.

Table 17. Buffer Control Actions

Action	Description	Syntax
`trigger`	Stops the acquisition for the current buffer and ends analysis. This command is required in every flow definition.	`trigger <`*`post-fill_count`*`>;`
`segment_trigger`	Available only in segmented acquisition mode. Ends acquisition of the current segment. After evaluating this command, the Signal Tap logic analyzer starts acquiring from the next segment. If all segments are written, the logic analyzer overwrites the oldest segment with the latest sample. When a trigger action is evaluated the acquisition stops.	`segment_trigger <`*`post-fill_count`*`>;`
`start_store`	Active only in state-based storage qualifier mode. Asserts the `write_enable` to the Signal Tap acquisition buffer.	`start_store`
`stop_store`	Active only in state-based storage qualifier mode. De-asserts the `write_enable` signal to the Signal Tap acquisition buffer.	`stop_store`

Both `trigger` and `segment_trigger` actions accept an optional `post-fill_count` argument.

State Transition Action

Specifies the next state in the custom state control flow. The syntax is:
`goto <`*`state_label`*`>;`

Trigger that Skips Clock Cycles after Hitting Condition

Trigger flow description that skips three clock cycles of samples after hitting condition 1

Code:

```
State 1: ST1
    start_store
    if ( condition1 )
    begin
       stop_store;
       goto ST2;
    end
State 2: ST2
    if (c1 < 3)
       increment c1; //skip three clock cycles; c1 initialized to 0
    else if (c1 == 3)
    begin
       start_store;//start_store necessary to enable writing to finish
                    //acquisition
       trigger;
    end
```

The figures show the data transaction on a continuous capture and the data capture when you apply the Trigger flow description.

Figure 56. Continuous Capture of Data Transaction

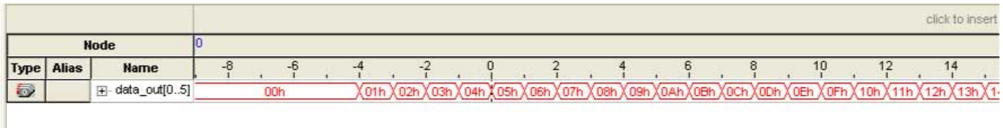
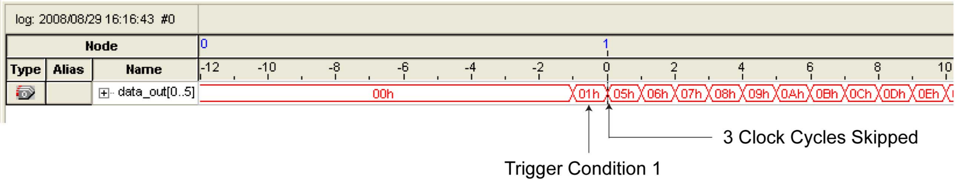


Figure 57. Capture of Data Transaction with Trigger Flow Description Applied

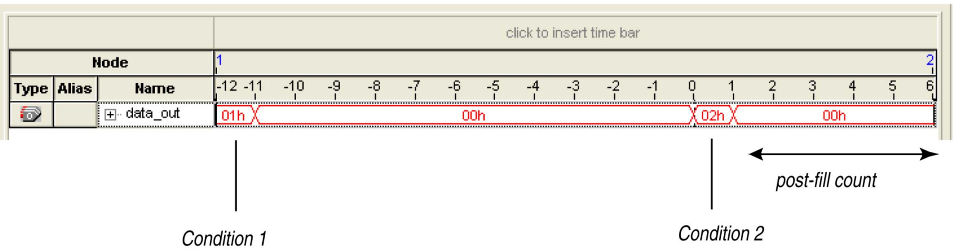


Storage Qualification with Post-Fill Count Value Less than m

The data capture finishes successfully. It uses a buffer with a sample depth of 64, *m = n = 10*, and `post-fill count = 5`.

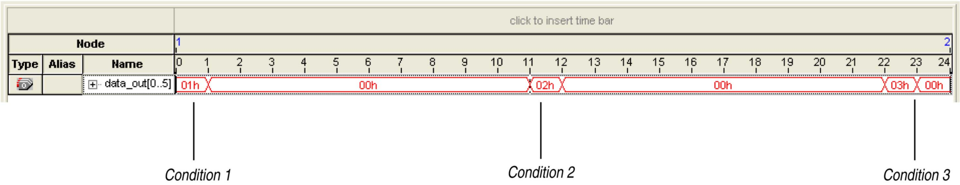
Real data acquisition of the previous scenario

Figure 58. Storage Qualification with Post-Fill Count Value Less than m (Acquisition Successfully Completes)



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.

Figure 59. Waveform After Forcing the Analysis to Stop



Resource Manipulation Action

The resources the trigger flow description uses can be either counters or status flags.

Table 18. Resource Manipulation Actions

Action	Description	Syntax
increment	Increments a counter resource by 1	`increment <counter_identifier>;`
decrement	Decrements a counter resource by 1	`decrement <counter_identifier>;`
reset	Resets counter resource to initial value	`reset <counter_identifier>;`
set	Sets a status flag to 1	`set <register_flag_identifier>;`
clear	Sets a status flag to 0	`clear <register_flag_identifier>;`

Buffer Control Actions

Actions that control the acquisition buffer.

Table 19. Buffer Control Actions

Action	Description	Syntax
trigger	Stops the acquisition for the current buffer and ends analysis. This command is required in every flow definition.	`trigger <post-fill_count>;`
segment_trigger	Available only in segmented acquisition mode. Ends acquisition of the current segment. After evaluating this command, the Signal Tap logic analyzer starts acquiring from the next segment. If all segments are written, the logic analyzer overwrites the oldest segment with the latest sample. When a trigger action is evaluated the acquisition stops.	`segment_trigger <post-fill_count>;`
start_store	Active only in state-based storage qualifier mode. Asserts the write_enable to the Signal Tap acquisition buffer.	`start_store`
stop_store	Active only in state-based storage qualifier mode. De-asserts the write_enable signal to the Signal Tap acquisition buffer.	`stop_store`

Both trigger and segment_trigger actions accept an optional post-fill_count argument.

Related Information

Post-fill Count on page 59

State Transition Action

Specifies the next state in the custom state control flow. The syntax is:
goto *<state_label>*;

2.4.6.11.6. State-Based Storage Qualifier Feature

Selecting a state-based storage qualifier type enables the `start_store` and `stop_store` actions. When you use these actions in conjunction with the expressions of the State-based trigger flow, you get maximum flexibility to control data written into the acquisition buffer.

Note: You can only apply the `start_store` and `stop_store` commands to a non-segmented buffer.

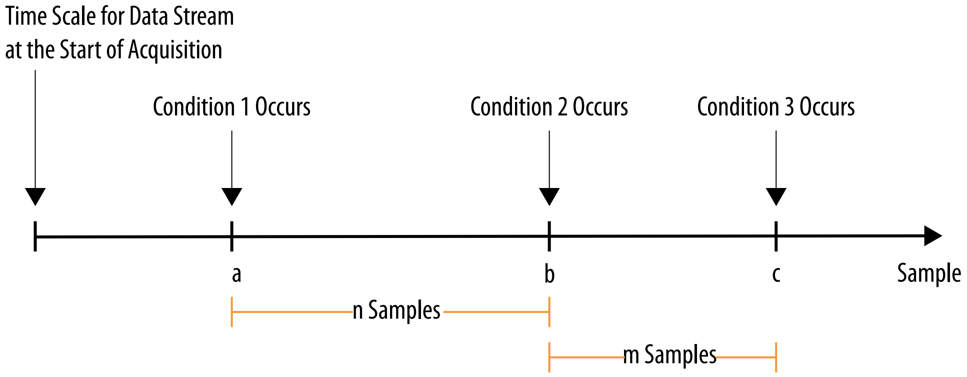
The `start_store` and `stop_store` commands are similar to the start and stop conditions of the **start/stop** storage qualifier mode. If you enable storage qualification, the Signal Tap logic analyzer doesn't write data into the acquisition buffer until the `start_store` command occurs. However, in the state-based storage qualifier type you must include a `trigger` command as part of the trigger flow description. This `trigger` command is necessary to complete the acquisition and display the results on the waveform display.

Storage Qualification Feature for the State-Based Trigger Flow

This trigger flow description contains three trigger conditions that occur at different times after you click **Start Analysis**:

```
State 1: ST1:
    if ( condition1 )
       start_store;
    else if ( condition2 )
       trigger value;
    else if ( condition3 )
       stop_store;
```

Figure 60. Capture Scenario for Storage Qualification with the State-Based Trigger Flow



When you apply the trigger flow to the scenario in the figure:

1. The Signal Tap logic analyzer does not write into the acquisition buffer until **Condition 1** occurs (sample **a**).
2. When **Condition 2** occurs (sample **b**), the logic analyzer evaluates the `trigger value` command, and continues to write into the buffer to finish the acquisition.
3. The trigger flow specifies a `stop_store` command at sample **c**, which occurs `m` samples after the trigger point.
4. If the data acquisition finishes the post-fill acquisition samples before **Condition 3** occurs, the logic analyzer finishes the acquisition and displays the contents of the waveform. In this case, the capture ends if the post-fill count value is < `m`.
5. If the post-fill count value in the Trigger Flow description 1 is > `m` samples, the buffer pauses acquisition indefinitely, provided there is no recurrence of Condition 1 to trigger the logic analyzer to start capturing data again.

The Signal Tap logic analyzer continues to evaluate the `stop_store` and `start_store` commands even after evaluating the trigger. If the acquisition paused, click **Stop Analysis** to manually stop and force the acquisition to trigger. You can use counter values, flags, and the State diagram to help you perform the trigger flow. The counter values, flags, and the current state update in real-time during a data acquisition.

2.4.6.12. Trigger Lock Mode

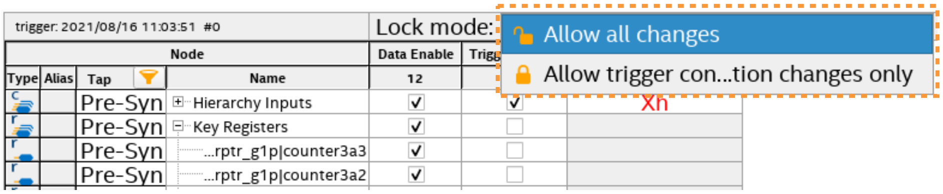
Trigger lock mode restricts changes to only the configuration settings that you specify as **Configurable at runtime**. The runtime configurable settings for the **Custom Trigger Flow** tab are on by default.

Note: You may get some performance advantages by disabling some of the runtime configurable options.

You can restrict changes to your Signal Tap configuration to include only the options that do not require a recompilation. Trigger lock-mode allows you to make changes that reflect immediately in the device.

1. On the **Setup** tab, point to **Lock mode** and select **Allow trigger condition changes only**.

Figure 61. Allow Trigger Conditions Change Only



trigger: 2021/08/16 11:03:51 #0

Type	Alias	Tap	Name	Data Enable 12	Trigg	
c		Pre-Syn	⊞ Hierarchy Inputs	☑	☑	Xh
r		Pre-Syn	⊟ Key Registers	☑	□	
r		Pre-Syn	...rptr_g1p|counter3a3	☑	□	
r		Pre-Syn	...rptr_g1p|counter3a2	☑	□	

2. Modify the Trigger Flow conditions.

2.4.7. Specifying Pipeline Settings

The **Pipeline factor** setting indicates the number of pipeline registers that the Intel Quartus Prime software can add to boost the f_{MAX} of the Signal Tap logic analyzer.

To specify the pipeline factor from the Signal Tap GUI:

- In the **Signal Configuration** pane, specify a **pipeline factor** ranging from 0 to 5. The default value is 0.

Note: Setting the pipeline factor does not guarantee an increase in f_{MAX}, as the pipeline registers may not be in the critical paths.

Alternatively, you can specify pipeline parameters as part of HDL instantiation, as Creating a Signal Tap Instance by HDL Instantiation on page 34 describes.

Note: The Signal Tap Intel FPGA IP is not optimized for the Intel Hyperflex architecture.

2.4.8. Filtering Relevant Samples

The Storage Qualifier feature allows you to filter out individual samples not relevant to debugging your design.

The Signal Tap logic analyzer offers a snapshot in time of the data that the acquisition buffers store. By default, the Signal Tap logic analyzer writes into acquisition memory with data samples on every clock cycle. With a non-segmented buffer, there is one data window that represents a comprehensive snapshot of the data stream. Conversely, segmented buffers use several smaller sampling windows spread out over more time, with each sampling window representing a contiguous data set.

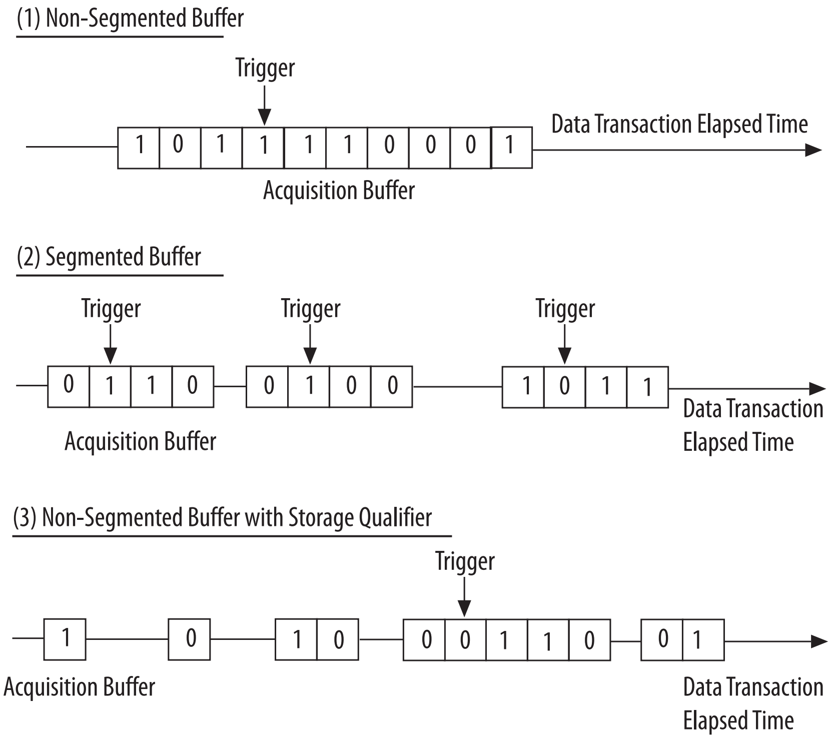
With analysis using acquisition buffers you can capture most functional errors in a chosen signal set, provided adequate trigger conditions and a generous sample depth for the acquisition. However, each data window can have a considerable amount of unnecessary data; for example, long periods of idle signals between data bursts. The default behavior in the Signal Tap logic analyzer doesn't discard the redundant sample bits.

The Storage Qualifier feature allows you to establish a condition that acts as a write enable to the buffer during each clock cycle of data acquisition, thus allowing a more efficient use of acquisition memory over a longer period of analysis.

Because you can create a discontinuity between any two samples in the buffer, the Storage Qualifier feature is equivalent to creating a custom segmented buffer in which the number and size of segment boundaries are adjustable.

Note: You can only use the Storage Qualifier feature with a non-segmented buffer. The IP Catalog flow only supports the Input Port mode for the Storage Qualifier feature.

Figure 62. Data Acquisition Using Different Modes of Controlling the Acquisition Buffer



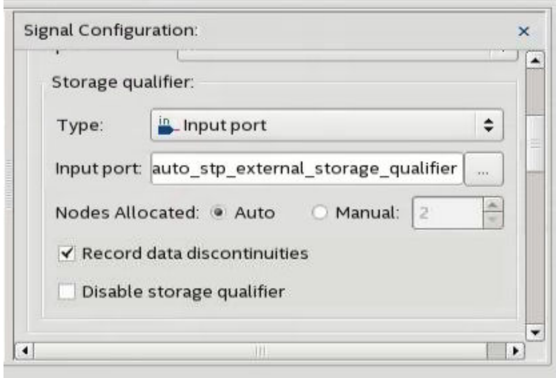
Notes to figure:

1. Non-segmented buffers capture a fixed sample window of contiguous data.
2. Segmented buffers divide the buffer into fixed sized segments, with each segment having an equal sample depth.
3. Storage Qualifier allows you to define a custom sampling window for each segment you create with a qualifying condition, thus potentially allowing a larger time scale of coverage.

There are six storage qualifier types available under the Storage Qualifier feature:

- **Continuous** (default) Turns the Storage Qualifier off.
- **Input port**
- **Transitional**
- **Conditional**
- **Start/Stop**
- **State-based**

Figure 63. Storage Qualifier Settings



Signal Configuration:

Storage qualifier:

Type: Input port

Input port: auto_stp_external_storage_qualifier

Nodes Allocated: ● Auto ○ Manual: 2

☑ Record data discontinuities

□ Disable storage qualifier

Upon the start of an acquisition, the Signal Tap logic analyzer examines each clock cycle and writes the data into the buffer based upon the storage qualifier type and condition. Acquisition stops when a defined set of trigger conditions occur.

The Signal Tap logic analyzer evaluates trigger conditions independently of storage qualifier conditions.

2.4.8.1. Input Port Mode

When using the Input port mode, the Signal Tap logic analyzer takes any signal from your design as an input. During acquisition, if the signal is high on the clock edge, the Signal Tap logic analyzer stores the data in the buffer. If the signal is low on the clock edge, the logic analyzer ignores the data sample. If you don't specify an internal node, the logic analyzer creates and connects a pin to this input port.

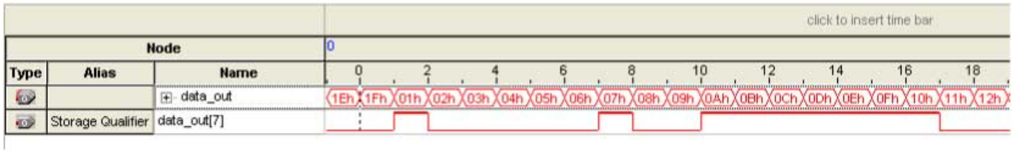
When creating a Signal Tap logic analyzer instance with the Signal Tap logic analyzer GUI, specify the **Storage Qualifier** signal for the **Input port** field located on the **Setup** tab. You must specify this port for your project to compile.

When creating a Signal Tap logic analyzer instance through HDL instantiation, specify the **Storage Qualifier** parameter to include in the instantiation template. You can then connect this port to a signal in your RTL. If you enable the input port storage qualifier, the port accepts a signal and predicates when signals are recorded into the acquisition buffer before or after the specified trigger condition occurs. That is, the trigger you specify is responsible for triggering and moving the logic analyzer into the post-fill state. The input port storage qualifier signal you select controls the recording of samples.

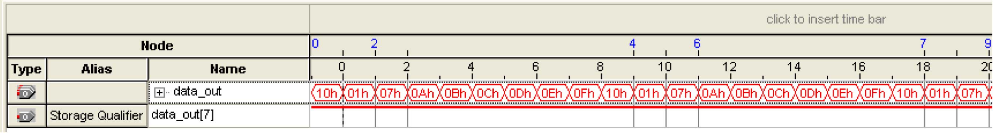
The following example compares and contrasts two waveforms of the same data, one without storage qualifier enabled (**Continuous** means always record samples, effectively no storage qualifier), and the other with **Input Port** mode. The bottom signal in the waveform, `data_out[7]`,is the input port storage qualifier signal. The continuous mode waveform shows 01h, 07h, 0Ah, 0Bh, 0Ch, 0Dh, 0Eh, 0Fh, 10h as the sequence of `data_out[7]` bus values where the storage qualifier signal is asserted. The lower waveform for input port storage qualifier shows how this same traffic pattern of the `data_out` bus is recorded when you enable the input port storage qualifier. Values recorded are a repeating sequence of the 01h, 07h, 0Ah, 0Bh, 0Ch, 0Dh, 0Eh, 0Fh, 10h (same as **Continuous** mode).

Figure 64. Comparing Continuous and Input Port Capture Mode in Data Acquisition of a Recurring Data Pattern

- **Continuous** Mode:



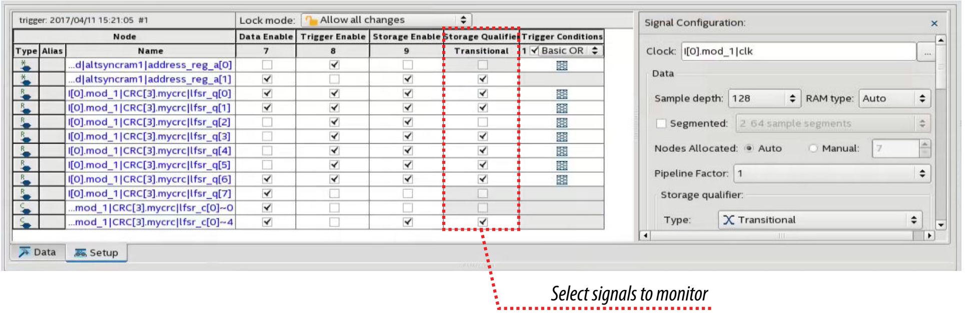
- **Input Port** Storage Qualifier:



2.4.8.2. Transitional Mode

In **Transitional** mode, the logic analyzer monitors changes in a set of signals, and writes new data in the acquisition buffer only after detecting a change. You select the signals for monitoring using the check boxes in the **Storage Qualifier** column.

Figure 65. Transitional Storage Qualifier Setup



trigger: 2017/04/11 15:21:05 #1 | Lock mode: Allow all changes

Node			Data Enable	Trigger Enable	Storage Enable	Storage Qualifier	Trigger Conditions
Type	Alias	Name	7	8	9	Transitional	1 ✓ Basic OR
		...d|altsyncram1|address_reg_a[0]	□	☑	□	□	▨
		...d|altsyncram1|address_reg_a[1]	☑	□	☑	☑	
		I[0].mod_1|CRC[3].mycrc|lfsr_q[0]	☑	☑	☑	☑	▨
		I[0].mod_1|CRC[3].mycrc|lfsr_q[1]	☑	☑	☑	☑	▨
		I[0].mod_1|CRC[3].mycrc|lfsr_q[2]	□	☑	☑	□	▨
		I[0].mod_1|CRC[3].mycrc|lfsr_q[3]	□	☑	☑	☑	▨
		I[0].mod_1|CRC[3].mycrc|lfsr_q[4]	□	☑	☑	☑	▨
		I[0].mod_1|CRC[3].mycrc|lfsr_q[5]	□	☑	☑	☑	▨
		I[0].mod_1|CRC[3].mycrc|lfsr_q[6]	☑	☑	☑	☑	▨
		I[0].mod_1|CRC[3].mycrc|lfsr_q[7]	☑	□	□	□	
		...mod_1|CRC[3].mycrc|lfsr_c[0]~0	☑	□	□	□	
		...mod_1|CRC[3].mycrc|lfsr_c[0]~4	☑	□	☑	☑	

Signal Configuration:

Clock: I[0].mod_1|clk

Data

Sample depth: 128 RAM type: Auto

□ Segmented: 2 64 sample segments

Nodes Allocated: ● Auto ○ Manual: 7

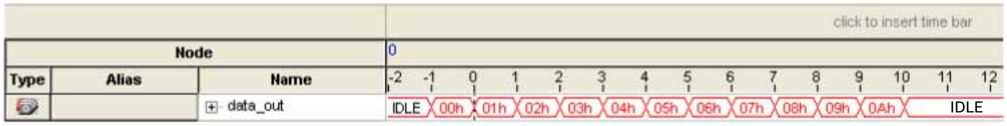
Pipeline Factor: 1

Storage qualifier:

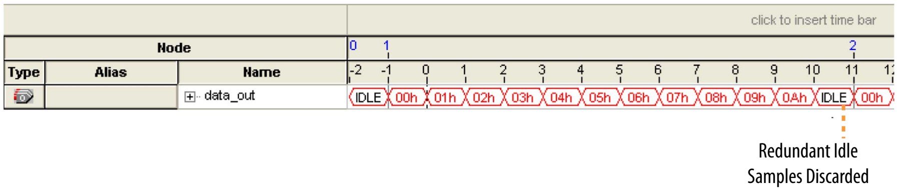
Type: Transitional

Figure 66. Comparing Continuous and Transitional Capture Mode in Data Acquisition of a Recurring Data Pattern

- **Continuous** mode:



- **Transitional** mode:


click to insert time bar
Node
Type
Alias
Name
0 1 2
-2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12
data_out
IDLE 00h 01h 02h 03h 04h 05h 06h 07h 08h 09h 0Ah IDLE 00h
Redundant Idle Samples Discarded

2.4.8.3. Conditional Mode

In **Conditional** mode, the Signal Tap logic analyzer determines whether to store a sample by evaluating a combinational function of predefined signals within the node list. The Signal Tap logic analyzer writes into the buffer during the clock cycles in which the condition you specify evaluates TRUE.

You can select either **Basic AND**, **Basic OR**, **Comparison**, or **Advanced** storage qualifier conditions. A **Basic AND** or **Basic OR** condition matches each signal to one of the following:

- **Don't Care**
- **Low**
- **High**
- **Falling Edge**
- **Rising Edge**
- **Either Edge**

If you specify a **Basic AND** storage qualifier condition for more than one signal, the Signal Tap logic analyzer evaluates the logical AND of the conditions.

You can specify any other combinational or relational operators with the enabled signal set for storage qualification through advanced storage conditions.

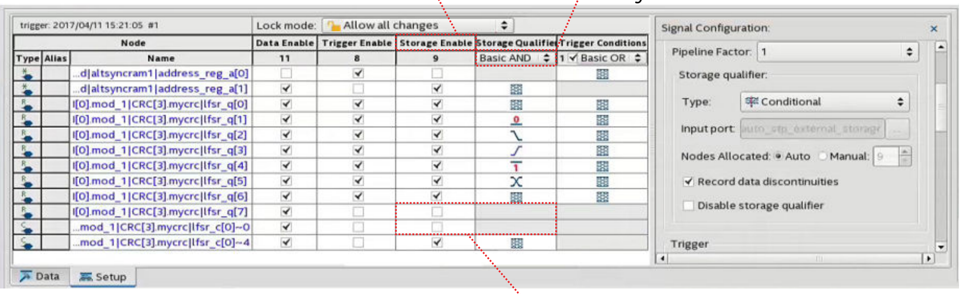
You can define storage qualification conditions similar to the manner in which you define trigger conditions.

Figure 67. Conditional Storage Qualifier Setup

The figure details the conditional storage qualifier setup in the `.stp` file.

Storage Enable

Storage Condition



trigger: 2017/04/11 15:21:05 #1 | Lock mode: Allow all changes

Type	Alias	Name	Data Enable 11	Trigger Enable 8	Storage Enable 9	Storage Qualifier Basic AND	Trigger Conditions 1 ✓ Basic OR
		...d|altsyncram1|address_reg_a[0]	□	☑	□		▦
		...d|altsyncram1|address_reg_a[1]	☑	□	☑	▦	
		l[0].mod_1|CRC[3].mycrc|lfsr_q[0]	☑	☑	☑	▦	▦
		l[0].mod_1|CRC[3].mycrc|lfsr_q[1]	☑	☑	☑	0	▦
		l[0].mod_1|CRC[3].mycrc|lfsr_q[2]	☑	☑	☑	╲	▦
		l[0].mod_1|CRC[3].mycrc|lfsr_q[3]	☑	☑	☑	╱	▦
		l[0].mod_1|CRC[3].mycrc|lfsr_q[4]	☑	☑	☑	1	▦
		l[0].mod_1|CRC[3].mycrc|lfsr_q[5]	☑	☑	☑	X	▦
		l[0].mod_1|CRC[3].mycrc|lfsr_q[6]	☑	☑	☑	▦	▦
		l[0].mod_1|CRC[3].mycrc|lfsr_q[7]	☑	□	□		
		...mod_1|CRC[3].mycrc|lfsr_c[0]~0	☑	□	□		
		...mod_1|CRC[3].mycrc|lfsr_c[0]~4	☑	□	☑	▦	

Signal Configuration:

Pipeline Factor: 1

Storage qualifier:

Type: Conditional

Input port: auto_stp_external_storage

Nodes Allocated: ● Auto ○ Manual: 9

☑ Record data discontinuities

□ Disable storage qualifier

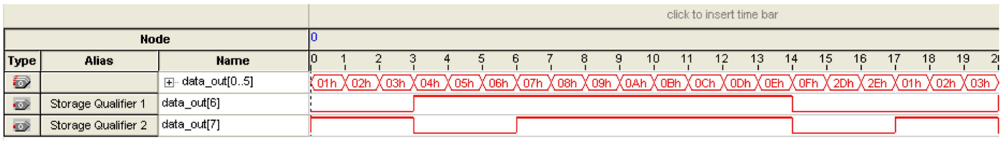
Trigger

Signals not enabled for storage cannot be part of the Storage Qualifier condition

Figure 68. Comparing Continuous and Conditional Capture Mode in Data Acquisition of a Recurring Data Pattern

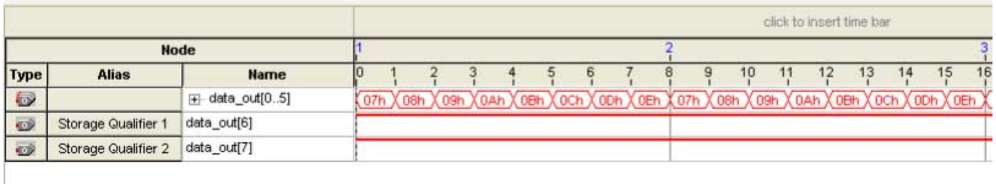
The data pattern is the same in both cases.

- **Continuous** sampling capture mode:



Type	Alias	Name
		⊞ data_out[0..5]
	Storage Qualifier 1	data_out[6]
	Storage Qualifier 2	data_out[7]

- **Conditional** sampling capture mode:



Type	Alias	Name
		⊞ data_out[0..5]
	Storage Qualifier 1	data_out[6]
	Storage Qualifier 2	data_out[7]

Related Information

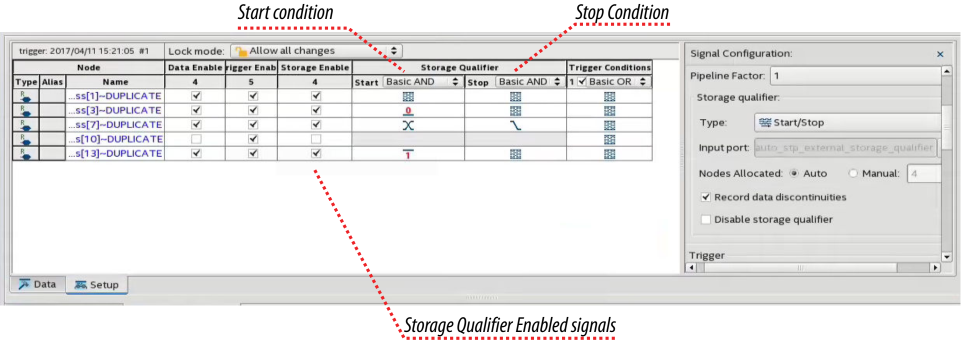
- Basic Trigger Conditions on page 50
- Comparison Trigger Conditions on page 51
- Advanced Trigger Conditions on page 53

2.4.8.4. Start/Stop Mode

The **Start/Stop** mode uses two sets of conditions, one to start data capture and one to stop data capture. If the start condition evaluates to TRUE, the Signal Tap logic analyzer stores the buffer data every clock cycle until the stop condition evaluates to TRUE, which then pauses the data capture. The logic analyzer ignores additional start signals received after the data capture starts. If both start and stop evaluate to TRUE at the same time, the logic analyzer captures a single cycle.

Note: You can force a trigger by pressing the **Stop** button if the buffer fails to fill to completion due to a stop condition or if the start condition never occurs.

Figure 69. Start/Stop Mode Storage Qualifier Setup



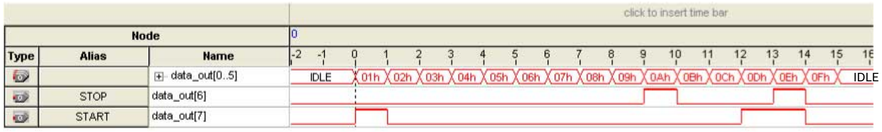
trigger: 2017/04/11 15:21:05 #1 Lock mode: Allow all changes

Node			Data Enable	rigger Enab	Storage Enable	Storage Qualifier		Trigger Conditions
Type	Alias	Name	4	5	4	Start Basic AND	Stop Basic AND	1 ✓ Basic OR
		...ss[1]~DUPLICATE	☑	☑	☑	▨	▨	▨
		...ss[3]~DUPLICATE	☑	☑	☑	0	▨	▨
		...ss[7]~DUPLICATE	☑	☑	☑	X	⌄	▨
		...s[10]~DUPLICATE	□	☑	□			▨
		...s[13]~DUPLICATE	☑	☑	☑	1	▨	▨

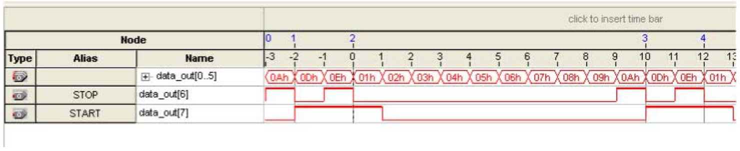
Signal Configuration:
Pipeline Factor: 1
Storage qualifier:
Type: Start/Stop
Input port: auto_stp_external_storage_qualifier
Nodes Allocated: ● Auto ○ Manual: 4
☑ Record data discontinuities
□ Disable storage qualifier
Trigger

Figure 70. Comparing Continuous and Start/Stop Acquisition Modes for a Recurring Data Pattern

- **Continuous** Mode:



- **Start/Stop** Storage Qualifier:



2.4.8.5. State-Based Mode

The State-based storage qualification mode is part of the State-based triggering flow. The state based triggering flow evaluates a conditional language to define how the Signal Tap logic analyzer writes data into the buffer. With the State-based trigger flow, you have command over boolean and relational operators to guide the execution flow for the target acquisition buffer.

When you enable the storage qualifier feature for the State-based flow, two additional commands become available: `start_store` and `stop_store`. These commands are similar to the Start/Stop capture conditions. Upon the start of acquisition, the Signal Tap logic analyzer doesn't write data into the buffer until a `start_store` action is performed. The `stop_store` command pauses the acquisition. If both `start_store` and `stop_store` actions occur within the same clock cycle, the logic analyzer stores a single sample into the acquisition buffer.

Related Information

State-Based Triggering on page 63

2.4.8.6. Showing Data Discontinuities

When you turn on **Record data discontinuities**, the Signal Tap logic analyzer marks the samples during which the acquisition paused from a storage qualifier. This marker is displayed in the waveform viewer after acquisition completes.

2.4.8.7. Disable the Storage Qualifier

You can disable the storage qualifier with the **Disable Storage Qualifier** option, and then perform a continuous capture. The **Disable Storage Qualifier** option is run-time reconfigurable. Changing the storage qualifier mode from the **Type** field requires recompilation of the project.

2.5. Step 3: Compile the Design and Signal Tap Instances

After you configure one or more Signal Tap instances and define trigger conditions, you must compile your project that includes the Signal Tap logic analyzer, prior to device configuration.

When you define a Signal Tap instance in the logic analyzer GUI or with HDL instantiation, the Signal Tap logic analyzer instance becomes part of your design for compilation.

To run full compilation of the design that includes the Signal Tap logic analyzer instance:

- Click **Processing ➤ Start Compilation**

You can employ various techniques to preserve specific signals for debugging during compilation, and to reduce overall compilation time and iterations. Refer to the following sections for more details.

2.5.1. Recompiling Only Signal Tap Changes

Certain Signal Tap configuration changes require a full recompilation of the design to implement. However, you can use the **Start Recompile** command to implement the following types of configuration changes without running a full design compilation.

Table 20. Signal Tap Configuration Changes Not Requiring Full Compilation

Change the post-fit tap target	Increase the number of post-fit targets
Change the post-fit tap inputs to a Basic AND trigger	Change the post-fit tap inputs to a Basic OR trigger
Change an Advanced trigger (post-fit inputs or logic)	Convert a pre-synthesis tap into a post-fit tap

Start Recompile appends Signal Tap node changes to the existing finalized snapshot, without changing placement and routing outside of the Signal Tap partition.

To recompile Signal Tap configuration changes only, follow these steps:

1. Make supported changes to the Signal Tap configuration in the **Signal Configuration** pane, according to Table 20 on page 80.
2. In the Signal Tap window, click **Processing ➤ Start Recompile**, or click the **Start Recompile** button. A dialog box displays whether each change is Supported or Unsupported by **Start Recompile**.

Figure 71. Signal Tap Toolbar Start Recompile Button and Command

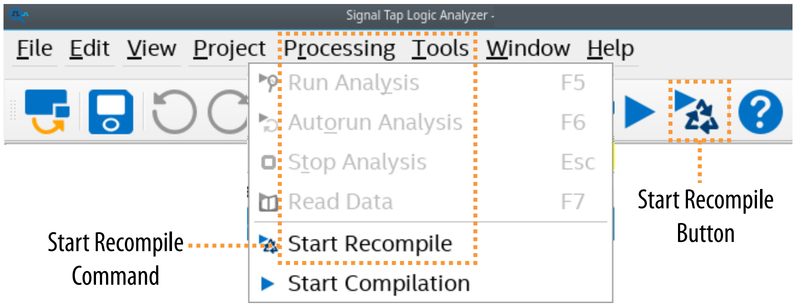
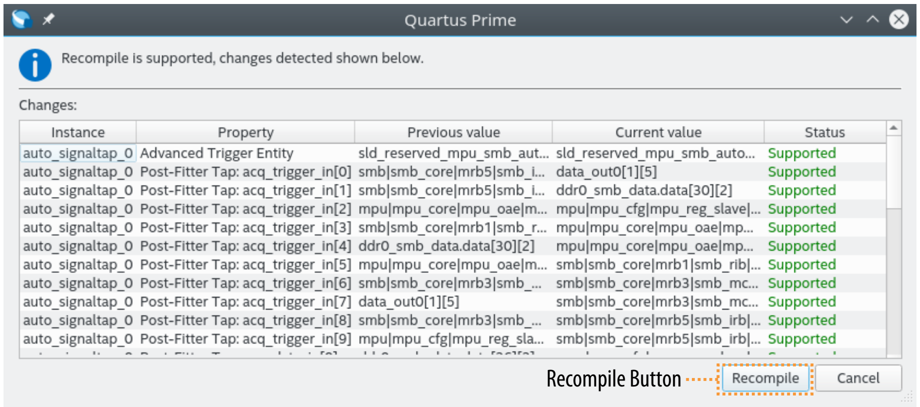


Figure 72. Recompilation Changes List



Recompile is supported, changes detected shown below.

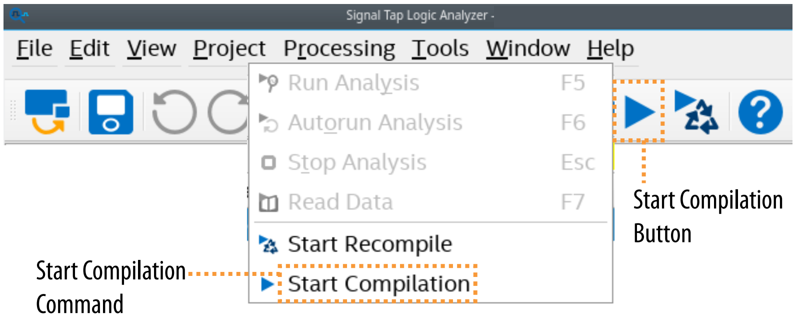
Changes:

Instance	Property	Previous value	Current value	Status
auto_signaltap_0	Advanced Trigger Entity	sld_reserved_mpu_smb_aut...	sld_reserved_mpu_smb_auto...	Supported
auto_signaltap_0	Post-Fitter Tap: acq_trigger_in[0]	smb|smb_core|mrb5|smb_i...	data_out0[1][5]	Supported
auto_signaltap_0	Post-Fitter Tap: acq_trigger_in[1]	smb|smb_core|mrb5|smb_i...	ddr0_smb_data.data[30][2]	Supported
auto_signaltap_0	Post-Fitter Tap: acq_trigger_in[2]	mpu|mpu_core|mpu_oae|m...	mpu|mpu_cfg|mpu_reg_slave|...	Supported
auto_signaltap_0	Post-Fitter Tap: acq_trigger_in[3]	smb|smb_core|mrb1|smb_r...	mpu|mpu_core|mpu_oae|mp...	Supported
auto_signaltap_0	Post-Fitter Tap: acq_trigger_in[4]	ddr0_smb_data.data[30][2]	mpu|mpu_core|mpu_oae|mp...	Supported
auto_signaltap_0	Post-Fitter Tap: acq_trigger_in[5]	mpu|mpu_core|mpu_oae|m...	smb|smb_core|mrb1|smb_rib|...	Supported
auto_signaltap_0	Post-Fitter Tap: acq_trigger_in[6]	smb|smb_core|mrb3|smb_...	smb|smb_core|mrb3|smb_mc...	Supported
auto_signaltap_0	Post-Fitter Tap: acq_trigger_in[7]	data_out0[1][5]	smb|smb_core|mrb3|smb_mc...	Supported
auto_signaltap_0	Post-Fitter Tap: acq_trigger_in[8]	smb|smb_core|mrb3|smb_...	smb|smb_core|mrb5|smb_irb|...	Supported
auto_signaltap_0	Post-Fitter Tap: acq_trigger_in[9]	mpu|mpu_cfg|mpu_reg_sla...	smb|smb_core|mrb5|smb_irb|...	Supported

3. If the Signal Tap configuration changes have a **Status** of Supported, click the **Recompile** button to recompile and implement only the Signal Tap configuration changes, as Figure 72 on page 81 shows.

4. For any change with **Status** of Unsupported, you must either revert the change to **Previous value**, or click **Processing ➤ Start Compilation** in Signal Tap to perform a full compilation to implement the change.

Figure 73. Signal Tap Toolbar Start Compilation Button and Command



Related Information

Changing the Post-Fit Target Nodes on page 84

2.5.2. Timing Preservation

The following techniques can help you preserve timing in designs that include the Signal Tap logic analyzer:

- Avoid adding critical path signals to the `.stp` file.
- Minimize the number of combinational signals you add to the `.stp` file, and add registers whenever possible.
- Specify an f_{MAX} constraint for each clock in the design.

Related Information

Timing Closure and Optimization
In *Intel Quartus Prime Pro Edition User Guide: Design Optimization*

2.5.3. Performance and Resource Considerations

When you perform logic analysis of your design, you can see the necessary trade-off between runtime flexibility, timing performance, and resource usage. The Signal Tap logic analyzer allows you to select runtime configurable parameters to balance the need for runtime flexibility, speed, and area.

The default values of the runtime configurable parameters provide maximum flexibility, so you can complete debugging as quickly as possible; however, you can adjust these settings to determine whether there is a more appropriate configuration for your design. Because performance results are design-dependent, try these options in different combinations until you achieve the desired balance between functionality, performance, and utilization.

2.5.3.1. Increasing Signal Tap Logic Performance

If Signal Tap logic is part of your critical path, follow these tips to speed up the performance of the Signal Tap logic:

- **Disable runtime configurable options**—runtime flexibility features expend some device resources. If you use Advanced Triggers or State-based triggering flow, disable runtime configurable parameters to a boost in f_{MAX} of the Signal Tap logic. If you use the State-based triggering flow, disable the **Goto state destination** option and perform a recompilation before disabling the other runtime configurable options. The **Goto state destination** option has the greatest impact on f_{MAX}, compared to the other runtime configurable options.
- **Minimize the number of signals that have Trigger Enable selected**—By default, the Signal Tap logic analyzer enables the **Trigger Enable** option for all signals that you add to the `.stp` file. For signals that you do not plan to use as triggers, turn this option off.
- **Turn on Physical Synthesis for register retiming**—If many (more than the number of inputs that fit in a LAB) enabled triggering signals fan-in logic to a gate-based triggering condition (basic trigger condition or a logical reduction operator in the advanced trigger tab), turn on **Perform register retiming**. This can help balance combinational logic across LABs.

2.5.3.2. Reducing Signal Tap Device Resources

If your design has resource constraints, follow these tips to reduce the logic or memory the Signal Tap logic analyzer requires:

- **Disable runtime configurable options**—disabling runtime configurability for advanced trigger conditions or runtime configurable options in the State-based triggering flow results in fewer LEs.
- **Minimize the number of segments in the acquisition buffer**—you can reduce the logic resources that the Signal Tap logic analyzer requires if you limit the segments in your sampling buffer.
- **Disable the Data Enable for signals that you use only for triggering**—by default, the Signal Tap logic analyzer enables **data enable** options for all signals. Turning off the **data enable** option for signals you use only as trigger inputs saves memory resources.

2.6. Step 4: Program the Target Hardware

After you add the Signal Tap logic analyzer instance to your project and fully compile the design, you configure the FPGA target device with your design that includes the Signal Tap logic analyzer instance. You can also program multiple devices with different designs and simultaneously debug them.

When you debug a design with the Signal Tap logic analyzer, you can program a target device directly using the supported JTAG hardware from the Signal Tap window, without using the Intel Quartus Prime Programmer.

Related Information

- Managing Multiple Signal Tap Configurations on page 99
- Intel Quartus Prime Pro Edition User Guide: Programmer

2.6.1. Ensure Compatibility Between .stp and .sof Files

The `.stp` file is compatible with a `.sof` file if the logic analyzer instance parameters, such as the size of the capture buffer and the monitoring and triggering signals, match the programming settings for the target device.

If the files are not compatible, you can still program the device, but you cannot run or control the logic analyzer from the Signal Tap logic analyzer GUI.

Use either of the following methods to ensure compatibility between `.stp` and `.sof` files

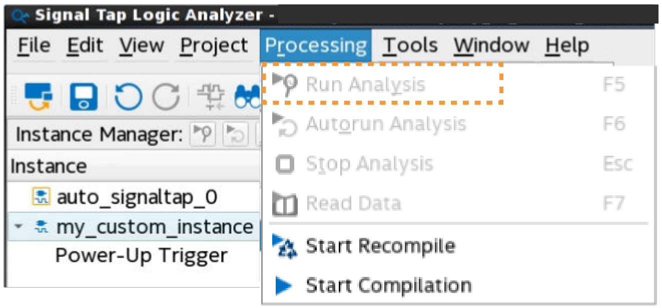
- Attach the .sof file to the .stp file in the SOF Manager. The SOF Manager ensures compatibility between any attached `.sof` files and the current `.stp` file settings automatically, as SOF Manager on page 100 describes.
- To ensure programming compatibility, program the FPGA device with the most recent `.sof` file.

Note: When the Signal Tap logic analyzer detects incompatibility after the analysis starts, the Intel Quartus Prime software generates a system error message containing two CRC values: the expected value and the value retrieved from the `.stp` instance on the device. The CRC value comes from all Signal Tap settings that affect the compilation.

2.7. Step 5: Run the Signal Tap Logic Analyzer

Debugging signals with the Signal Tap logic analyzer GUI is similar to debugging with an external logic analyzer. During normal device operation, you control the logic analyzer through the JTAG connection, specifying the start time for trigger conditions to begin capturing data.

Figure 74. Starting Signal Tap Analysis



1. Select the Signal Tap instance, and then initialize the logic analyzer for that instance by clicking **Processing ➤ Run Analysis** in the Signal Tap logic analyzer GUI.
2. When a trigger event occurs, the logic analyzer stores the captured data in the FPGA device's memory buffer, and then transfers this data to the **Signal Configuration** pane **Data** tab. You can perform the equivalent of a force trigger instruction that allows you to view the captured data currently in the buffer without a trigger event occurring.

You can also use In-System Sources and Probes in conjunction with the Signal Tap logic analyzer to force trigger conditions. The In-System Sources and Probes feature allows you to drive and sample values on to selected signals over the JTAG chain.

2.7.1. Changing the Post-Fit Signal Tap Target Nodes

After performing full compilation of your design and Signal Tap instance, you can subsequently make iterative changes to the post-fit Signal Tap nodes that you want to target, without rerunning full compilation to implement the changes.

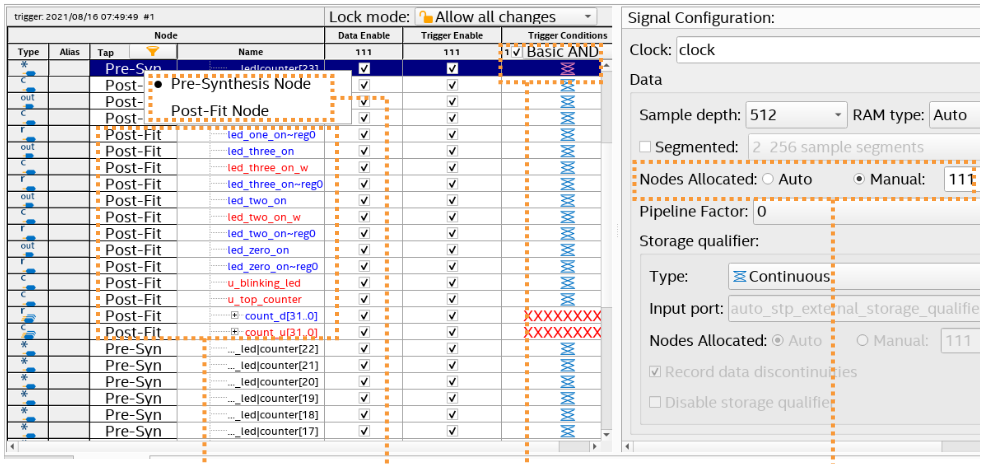
The Signal Tap **Node** list displays whether a target node is **Pre-Syn** (pre-synthesis) or **Post-Fit** in the filterable **Tap** column.

To modify the post-fit Signal Tap nodes:

1. Optionally, mark signals for debug, as Preserving Signals for Monitoring and Debugging on page 39 describes.

   Note: You cannot change all pre-synthesis nodes to post-fit nodes, unless you are changing the nodes before running full compilation. Once you preserve any signal with `preserve_for_debug`, you can change those preserved pre-synthesis nodes to post-fit nodes.
2. In the **Signal Configuration** pane, modify any of the following properties for nodes with a **Tap** of **Post-Fit**:

Figure 75. Changing the Post-Fit Signal Tap Nodes



trigger: 2021/08/16 07:49:49 #1 | Lock mode: Allow all changes

Node				Data Enable	Trigger Enable	Trigger Conditions
Type	Alias	Tap	Name	111	111	1 ☑ Basic AND
*		Pre-Syn	led|counter[23]	☑	☑	⧖
c		Post-		☑	☑	⧖
out		Post-		☑	☑	⧖
c		Post-		☑	☑	⧖
r		Post-Fit	led_one_on~reg0	☑	☑	⧖
out		Post-Fit	led_three_on	☑	☑	⧖
c		Post-Fit	led_three_on_w	☑	☑	⧖
r		Post-Fit	led_three_on~reg0	☑	☑	⧖
out		Post-Fit	led_two_on	☑	☑	⧖
c		Post-Fit	led_two_on_w	☑	☑	⧖
r		Post-Fit	led_two_on~reg0	☑	☑	⧖
out		Post-Fit	led_zero_on	☑	☑	⧖
r		Post-Fit	led_zero_on~reg0	☑	☑	⧖
c		Post-Fit	u_blinking_led	☑	☑	⧖
c		Post-Fit	u_top_counter	☑	☑	⧖
r		Post-Fit	⊞ count_d[31..0]	☑	☑	XXXXXXXX
c		Post-Fit	⊞ count_u[31..0]	☑	☑	XXXXXXXX
*		Pre-Syn	..._led|counter[22]	☑	☑	⧖
*		Pre-Syn	..._led|counter[21]	☑	☑	⧖
*		Pre-Syn	..._led|counter[20]	☑	☑	⧖
*		Pre-Syn	..._led|counter[19]	☑	☑	⧖
*		Pre-Syn	..._led|counter[18]	☑	☑	⧖
*		Pre-Syn	..._led|counter[17]	☑	☑	⧖

Signal Configuration:

Clock: clock

Data

Sample depth: 512 RAM type: Auto

□ Segmented: 2 256 sample segments

Nodes Allocated: ○ Auto ● Manual: 111

Pipeline Factor: 0

Storage qualifier:

Type: Continuous

Input port: auto_stp_external_storage_qualifie

Nodes Allocated: ● Auto ○ Manual: 111

☑ Record data discontinuities

□ Disable storage qualifier

Change Post-Fit Targets

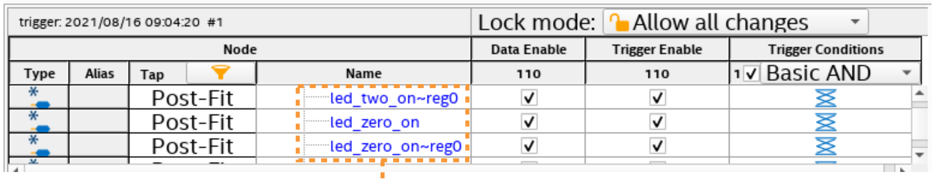
Convert Pre-Synthesis to Post-Fit Nodes

Change Trigger Mode

Change Number of Nodes

- In the **Name** column, modify or add a new post-fit Signal Tap node target, regardless of the trigger mode.
- In the **Trigger Conditions** column, modify the trigger mode. You can add the post-fit Signal Tap node inputs to a **Basic AND** or **Basic OR** trigger.
- In **Nodes Allocated**, you can specify the **Manual** option to increase or decrease the number of post-fit node targets. You can use manual allocation to help you avoid any major logic change that may require a full recompilation. The data input width affects memory use. The trigger input and storage input width affects the complexity of the condition logic, which can increase the device resource use and the complexity of timing closure.
- Right-click any pre-synthesis Signal Tap node to convert to a post-fit Signal Tap node. The conversion is only successful if Signal Tap can resolve pre-synthesis to post-fit name mapping. Otherwise, the node appears in red and connected to ground. When conversion is successful the post-fit taps names appear in blue text.

Figure 76. Post-fit Taps Names Appear in Blue Text



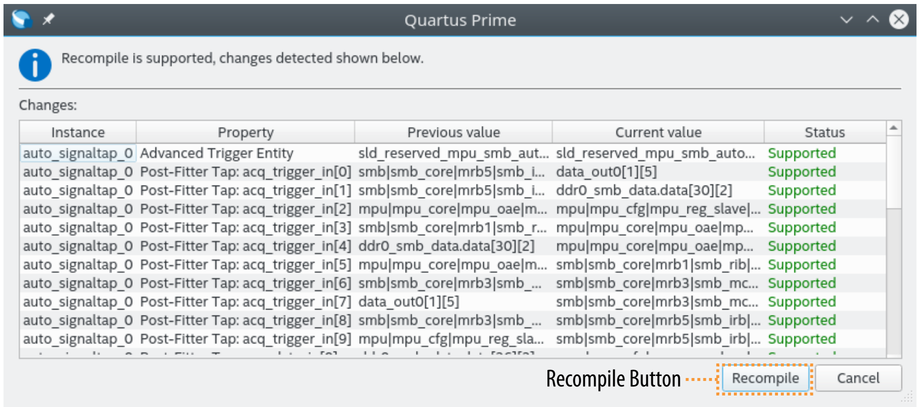
trigger: 2021/08/16 09:04:20 #1 | Lock mode: Allow all changes

Node				Data Enable	Trigger Enable	Trigger Conditions
Type	Alias	Tap	Name	110	110	1 ☑ Basic AND
*		Post-Fit	led_two_on~reg0	☑	☑	⧖
*		Post-Fit	led_zero_on	☑	☑	⧖
*		Post-Fit	led_zero_on~reg0	☑	☑	⧖

Post-fit Nodes Appear in Blue Text

3. After your post-fit node changes are complete, click **Processing ➤ Start Recompile** to implement only the Signal Tap node changes. A dialog box appears that lists the changes you are implementing, and whether recompilation supports the change.

Figure 77. Recompilation Changes List



Recompile is supported, changes detected shown below.

Changes:

<table>
<tr><th>Instance</th><th>Property</th><th>Previous value</th><th>Current value</th><th>Status</th></tr>
<tr><td>auto_signaltap_0</td><td>Advanced Trigger Entity</td><td>sld_reserved_mpu_smb_aut...</td><td>sld_reserved_mpu_smb_auto...</td><td>Supported</td></tr>
<tr><td>auto_signaltap_0</td><td>Post-Fitter Tap: acq_trigger_in[0]</td><td>smb|smb_core|mrb5|smb_i...</td><td>data_out0[1][5]</td><td>Supported</td></tr>
<tr><td>auto_signaltap_0</td><td>Post-Fitter Tap: acq_trigger_in[1]</td><td>smb|smb_core|mrb5|smb_i...</td><td>ddr0_smb_data.data[30][2]</td><td>Supported</td></tr>
<tr><td>auto_signaltap_0</td><td>Post-Fitter Tap: acq_trigger_in[2]</td><td>mpu|mpu_core|mpu_oae|m...</td><td>mpu|mpu_cfg|mpu_reg_slave|...</td><td>Supported</td></tr>
<tr><td>auto_signaltap_0</td><td>Post-Fitter Tap: acq_trigger_in[3]</td><td>smb|smb_core|mrb1|smb_r...</td><td>mpu|mpu_core|mpu_oae|mp...</td><td>Supported</td></tr>
<tr><td>auto_signaltap_0</td><td>Post-Fitter Tap: acq_trigger_in[4]</td><td>ddr0_smb_data.data[30][2]</td><td>mpu|mpu_core|mpu_oae|mp...</td><td>Supported</td></tr>
<tr><td>auto_signaltap_0</td><td>Post-Fitter Tap: acq_trigger_in[5]</td><td>mpu|mpu_core|mpu_oae|m...</td><td>smb|smb_core|mrb1|smb_rib|...</td><td>Supported</td></tr>
<tr><td>auto_signaltap_0</td><td>Post-Fitter Tap: acq_trigger_in[6]</td><td>smb|smb_core|mrb3|smb_...</td><td>smb|smb_core|mrb3|smb_mc...</td><td>Supported</td></tr>
<tr><td>auto_signaltap_0</td><td>Post-Fitter Tap: acq_trigger_in[7]</td><td>data_out0[1][5]</td><td>smb|smb_core|mrb3|smb_mc...</td><td>Supported</td></tr>
<tr><td>auto_signaltap_0</td><td>Post-Fitter Tap: acq_trigger_in[8]</td><td>smb|smb_core|mrb3|smb_...</td><td>smb|smb_core|mrb5|smb_irb|...</td><td>Supported</td></tr>
<tr><td>auto_signaltap_0</td><td>Post-Fitter Tap: acq_trigger_in[9]</td><td>mpu|mpu_cfg|mpu_reg_sla...</td><td>smb|smb_core|mrb5|smb_irb|...</td><td>Supported</td></tr>
</table>

4. For any change with **Status** of **Unsupported**, you must either revert the change to **Previous value**, or perform a full compilation to implement the change.

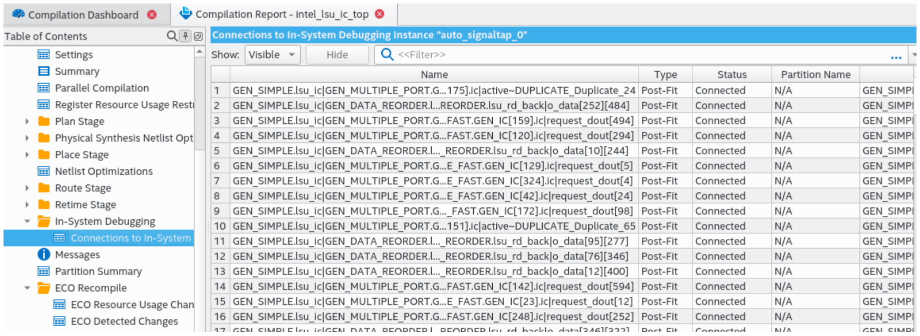
5. Click the **Recompile** button, as Figure 77 on page 86 shows. Recompilation uses the Engineering Change Order (ECO) compilation flow to append your Signal Tap node changes to the existing finalized snapshot, without changing placement and routing outside the Signal Tap partition.

   Note: The recompilation only applies to the project database if the recompilation is successful. Otherwise, the last successful compilation results remain unchanged.

6. View the changes in the following Compilation Reports following recompilation:

Figure 78. Connections to In-System Debugging Report

Lists each tap target and whether the connection successfully routes (is Connected after recompilation)

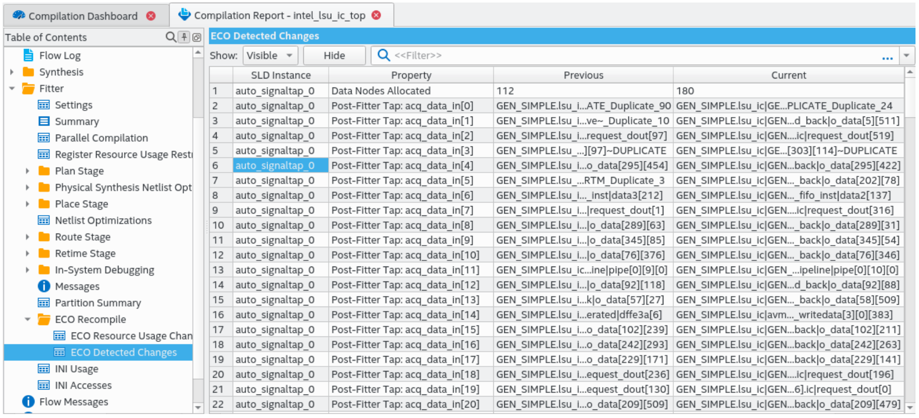


Connections to In-System Debugging Instance "auto_signaltap_0"

<table>
<tr><th></th><th>Name</th><th>Type</th><th>Status</th><th>Partition Name</th><th></th></tr>
<tr><td>1</td><td>GEN_SIMPLE.lsu_ic|GEN_MULTIPLE_PORT.G...175].ic|active~DUPLICATE_Duplicate_24</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
<tr><td>2</td><td>GEN_SIMPLE.lsu_ic|GEN_DATA_REORDER.l...REORDER.lsu_rd_back|o_data[252][484]</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
<tr><td>3</td><td>GEN_SIMPLE.lsu_ic|GEN_MULTIPLE_PORT.G...FAST.GEN_IC[159].ic|request_dout[494]</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
<tr><td>4</td><td>GEN_SIMPLE.lsu_ic|GEN_MULTIPLE_PORT.G...FAST.GEN_IC[120].ic|request_dout[294]</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
<tr><td>5</td><td>GEN_SIMPLE.lsu_ic|GEN_DATA_REORDER.l..._REORDER.lsu_rd_back|o_data[10][244]</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
<tr><td>6</td><td>GEN_SIMPLE.lsu_ic|GEN_MULTIPLE_PORT.G...E_FAST.GEN_IC[129].ic|request_dout[5]</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
<tr><td>7</td><td>GEN_SIMPLE.lsu_ic|GEN_MULTIPLE_PORT.G...E_FAST.GEN_IC[324].ic|request_dout[4]</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
<tr><td>8</td><td>GEN_SIMPLE.lsu_ic|GEN_MULTIPLE_PORT.G...E_FAST.GEN_IC[42].ic|request_dout[24]</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
<tr><td>9</td><td>GEN_SIMPLE.lsu_ic|GEN_MULTIPLE_PORT.G..._FAST.GEN_IC[172].ic|request_dout[98]</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
<tr><td>10</td><td>GEN_SIMPLE.lsu_ic|GEN_MULTIPLE_PORT.G...151].ic|active~DUPLICATE_Duplicate_65</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
<tr><td>11</td><td>GEN_SIMPLE.lsu_ic|GEN_DATA_REORDER.l..._REORDER.lsu_rd_back|o_data[95][277]</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
<tr><td>12</td><td>GEN_SIMPLE.lsu_ic|GEN_DATA_REORDER.l..._REORDER.lsu_rd_back|o_data[76][346]</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
<tr><td>13</td><td>GEN_SIMPLE.lsu_ic|GEN_DATA_REORDER.l..._REORDER.lsu_rd_back|o_data[12][400]</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
<tr><td>14</td><td>GEN_SIMPLE.lsu_ic|GEN_MULTIPLE_PORT.G...FAST.GEN_IC[142].ic|request_dout[594]</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
<tr><td>15</td><td>GEN_SIMPLE.lsu_ic|GEN_MULTIPLE_PORT.G...E_FAST.GEN_IC[23].ic|request_dout[12]</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
<tr><td>16</td><td>GEN_SIMPLE.lsu_ic|GEN_MULTIPLE_PORT.G...FAST.GEN_IC[248].ic|request_dout[252]</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
<tr><td>17</td><td>GEN_SIMPLE.lsu_ic|GEN_DATA_REORDER.l...REORDER.lsu_rd_back|o_data[346][322]</td><td>Post-Fit</td><td>Connected</td><td>N/A</td><td>GEN_SIMPI</td></tr>
</table>

Figure 79. ECO Detected Changes Report

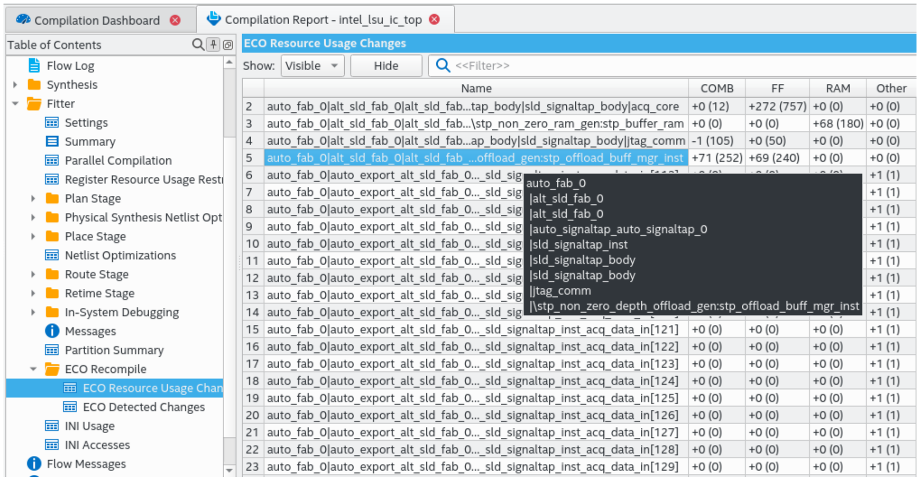
Lists each tap change that you implement with recompilation.



	SLD Instance	Property	Previous	Current
1	auto_signaltap_0	Data Nodes Allocated	112	180
2	auto_signaltap_0	Post-Fitter Tap: acq_data_in[0]	GEN_SIMPLE.lsu_i...ATE_Duplicate_90	GEN_SIMPLE.lsu_ic|GE...PLICATE_Duplicate_24
3	auto_signaltap_0	Post-Fitter Tap: acq_data_in[1]	GEN_SIMPLE.lsu_i...ve~_Duplicate_10	GEN_SIMPLE.lsu_ic|GEN...d_back|o_data[5][511]
4	auto_signaltap_0	Post-Fitter Tap: acq_data_in[2]	GEN_SIMPLE.lsu_i...request_dout[97]	GEN_SIMPLE.lsu_ic|GEN....ic|request_dout[519]
5	auto_signaltap_0	Post-Fitter Tap: acq_data_in[3]	GEN_SIMPLE.lsu_...][97]~DUPLICATE	GEN_SIMPLE.lsu_ic|GE...[303][114]~DUPLICATE
6	auto_signaltap_0	Post-Fitter Tap: acq_data_in[4]	GEN_SIMPLE.lsu_i...o_data[295][454]	GEN_SIMPLE.lsu_ic|GEN...back|o_data[295][422]
7	auto_signaltap_0	Post-Fitter Tap: acq_data_in[5]	GEN_SIMPLE.lsu_...RTM_Duplicate_3	GEN_SIMPLE.lsu_ic|GEN..._back|o_data[202][78]
8	auto_signaltap_0	Post-Fitter Tap: acq_data_in[6]	GEN_SIMPLE.lsu_i..._inst|data3[212]	GEN_SIMPLE.lsu_ic|GEN..._fifo_inst|data2[137]
9	auto_signaltap_0	Post-Fitter Tap: acq_data_in[7]	GEN_SIMPLE.lsu_i...|request_dout[1]	GEN_SIMPLE.lsu_ic|GEN....ic|request_dout[316]
10	auto_signaltap_0	Post-Fitter Tap: acq_data_in[8]	GEN_SIMPLE.lsu_i...|o_data[289][63]	GEN_SIMPLE.lsu_ic|GEN..._back|o_data[289][31]
11	auto_signaltap_0	Post-Fitter Tap: acq_data_in[9]	GEN_SIMPLE.lsu_i...|o_data[345][85]	GEN_SIMPLE.lsu_ic|GEN..._back|o_data[345][54]
12	auto_signaltap_0	Post-Fitter Tap: acq_data_in[10]	GEN_SIMPLE.lsu_i...|o_data[76][376]	GEN_SIMPLE.lsu_ic|GEN..._back|o_data[76][346]
13	auto_signaltap_0	Post-Fitter Tap: acq_data_in[11]	GEN_SIMPLE.lsu_ic...ine|pipe[0][9][0]	GEN_SIMPLE.lsu_ic|GEN_...ipeline|pipe[0][10][0]
14	auto_signaltap_0	Post-Fitter Tap: acq_data_in[12]	GEN_SIMPLE.lsu_i...|o_data[92][118]	GEN_SIMPLE.lsu_ic|GEN...d_back|o_data[92][88]
15	auto_signaltap_0	Post-Fitter Tap: acq_data_in[13]	GEN_SIMPLE.lsu_i...k|o_data[57][27]	GEN_SIMPLE.lsu_ic|GEN..._back|o_data[58][509]
16	auto_signaltap_0	Post-Fitter Tap: acq_data_in[14]	GEN_SIMPLE.lsu_i...erated|dffe3a[6]	GEN_SIMPLE.lsu_ic|avm..._writedata[3][0][383]
17	auto_signaltap_0	Post-Fitter Tap: acq_data_in[15]	GEN_SIMPLE.lsu_i...o_data[102][239]	GEN_SIMPLE.lsu_ic|GEN...back|o_data[102][211]
18	auto_signaltap_0	Post-Fitter Tap: acq_data_in[16]	GEN_SIMPLE.lsu_i...o_data[242][293]	GEN_SIMPLE.lsu_ic|GEN...back|o_data[242][263]
19	auto_signaltap_0	Post-Fitter Tap: acq_data_in[17]	GEN_SIMPLE.lsu_i...o_data[229][171]	GEN_SIMPLE.lsu_ic|GEN...back|o_data[229][141]
20	auto_signaltap_0	Post-Fitter Tap: acq_data_in[18]	GEN_SIMPLE.lsu_i...equest_dout[236]	GEN_SIMPLE.lsu_ic|GEN....ic|request_dout[196]
21	auto_signaltap_0	Post-Fitter Tap: acq_data_in[19]	GEN_SIMPLE.lsu_i...equest_dout[130]	GEN_SIMPLE.lsu_ic|GEN...6].ic|request_dout[0]
22	auto_signaltap_0	Post-Fitter Tap: acq_data_in[20]	GEN_SIMPLE.lsu_i...o_data[209][509]	GEN_SIMPLE.lsu_ic|GEN...back|o_data[209][479]

Figure 80. ECO Resource Usage Change

Shows the device resource area change that recompilation implements. Use this report to approximate whether additional changes to the Signal Tap configuration are likely to succeed in combination with the overall design utilization reports.



	Name	COMB	FF	RAM	Other
2	auto_fab_0|alt_sld_fab_0|alt_sld_fab...tap_body|sld_signaltap_body|acq_core	+0 (12)	+272 (757)	+0 (0)	+0 (0)
3	auto_fab_0|alt_sld_fab_0|alt_sld_fab...\stp_non_zero_ram_gen:stp_buffer_ram	+0 (0)	+0 (0)	+68 (180)	+0 (0)
4	auto_fab_0|alt_sld_fab_0|alt_sld_fab...ap_body|sld_signaltap_body|jtag_comm	-1 (105)	+0 (50)	+0 (0)	+0 (0)
5	auto_fab_0|alt_sld_fab_0|alt_sld_fab_...offload_gen:stp_offload_buff_mgr_inst	+71 (252)	+69 (240)	+0 (0)	+0 (0)
6	auto_fab_0|auto_export_alt_sld_fab_0..._sld_sig				+1 (1)
7	auto_fab_0|auto_export_alt_sld_fab_0..._sld_sig				+1 (1)
8	auto_fab_0|auto_export_alt_sld_fab_0..._sld_sig				+1 (1)
9	auto_fab_0|auto_export_alt_sld_fab_0..._sld_sig				+1 (1)
10	auto_fab_0|auto_export_alt_sld_fab_0..._sld_sig				+1 (1)
11	auto_fab_0|auto_export_alt_sld_fab_0..._sld_sig				+1 (1)
12	auto_fab_0|auto_export_alt_sld_fab_0..._sld_sig				+1 (1)
13	auto_fab_0|auto_export_alt_sld_fab_0..._sld_sig				+1 (1)
14	auto_fab_0|auto_export_alt_sld_fab_0..._sld_sig				+1 (1)
15	auto_fab_0|auto_export_alt_sld_fab_0..._sld_signaltap_inst_acq_data_in[121]	+0 (0)	+0 (0)	+0 (0)	+1 (1)
16	auto_fab_0|auto_export_alt_sld_fab_0..._sld_signaltap_inst_acq_data_in[122]	+0 (0)	+0 (0)	+0 (0)	+1 (1)
17	auto_fab_0|auto_export_alt_sld_fab_0..._sld_signaltap_inst_acq_data_in[123]	+0 (0)	+0 (0)	+0 (0)	+1 (1)
18	auto_fab_0|auto_export_alt_sld_fab_0..._sld_signaltap_inst_acq_data_in[124]	+0 (0)	+0 (0)	+0 (0)	+1 (1)
19	auto_fab_0|auto_export_alt_sld_fab_0..._sld_signaltap_inst_acq_data_in[125]	+0 (0)	+0 (0)	+0 (0)	+1 (1)
20	auto_fab_0|auto_export_alt_sld_fab_0..._sld_signaltap_inst_acq_data_in[126]	+0 (0)	+0 (0)	+0 (0)	+1 (1)
21	auto_fab_0|auto_export_alt_sld_fab_0..._sld_signaltap_inst_acq_data_in[127]	+0 (0)	+0 (0)	+0 (0)	+1 (1)
22	auto_fab_0|auto_export_alt_sld_fab_0..._sld_signaltap_inst_acq_data_in[128]	+0 (0)	+0 (0)	+0 (0)	+1 (1)
23	auto_fab_0|auto_export_alt_sld_fab_0..._sld_signaltap_inst_acq_data_in[129]	+0 (0)	+0 (0)	+0 (0)	+1 (1)

```
auto_fab_0
 |alt_sld_fab_0
 |alt_sld_fab_0
 |auto_signaltap_auto_signaltap_0
 |sld_signaltap_inst
 |sld_signaltap_body
 |sld_signaltap_body
 |jtag_comm
 |\stp_non_zero_depth_offload_gen:stp_offload_buff_mgr_inst
```

Related Information

- Preserving Signals for Monitoring and Debugging on page 39
- Recompiling Signal Tap Configuration Changes on page 80
- Using the ECO Compilation Flow chapter, Intel Quartus Prime Pro Edition User Guide: Design Optimization

2.7.2. Runtime Reconfigurable Options

When you use Runtime Trigger mode, you can change certain settings in the `.stp` without requiring recompilation of the design.

Table 21. Runtime Reconfigurable Features

Runtime Reconfigurable Setting	Description
Basic Trigger Conditions and Basic Storage Qualifier Conditions	Change without recompiling all signals that have the Trigger condition turned on to any basic trigger condition value
Comparison Trigger Conditions and Comparison Storage Qualifier Conditions	All the comparison operands, the comparison numeric values, and the interval bound values are runtime-configurable. You can also switch from Comparison to **Basic OR** trigger at runtime without recompiling.
Advanced Trigger Conditions and Advanced Storage Qualifier Conditions	Many operators include runtime configurable settings. For example, all comparison operators are runtime-configurable. Configurable settings appear with a white background in the block representation. This runtime reconfigurable option is turned on in the **Object Properties** dialog box.
Switching between a storage-qualified and a continuous acquisition	Within any storage-qualified mode, you can switch to continuous capture mode without recompiling the design. To enable this feature, turn on **disable storage qualifier**.
State-based trigger flow parameters	Refer to *Runtime Reconfigurable Settings, State-Based Triggering Flow*

Runtime Reconfigurable options can save time during the debugging cycle by allowing you to cover a wider possible range of events, without requiring design recompilation. You may experience a slight impact to the performance and logic utilization. You can turn off runtime re-configurability for advanced trigger conditions and the state-based trigger flow parameters, boosting performance and decreasing area utilization.

To configure the `.stp` file to prevent changes that normally require recompilation in the **Setup** tab, select the **Allow Trigger Condition changes only** lock mode above the node list.

This example illustrates a potential use case for Runtime Reconfigurable features, by providing a storage qualified enabled State-based trigger flow description, and showing how to modify the size of a capture window at runtime without a recompile. This example gives you equivalent functionality to a segmented buffer with a single trigger condition where the segment sizes are runtime reconfigurable.

```
state ST1:
if ( condition1 && (c1 <= m) )// each "segment"  triggers on condition // 1
begin                           // m  = number of total "segments"
   start_store;
   increment c1;
   goto ST2:
end

else (c1 > m )                // This else condition handles the last
                              // segment.
begin
    start_store
    trigger (n-1)
end

state ST2:
if ( c2 >= n)                //n = number of samples to capture in each
                             //segment.
begin
    reset c2;
    stop_store;
    goto ST1;
end

else (c2 < n)
begin
```

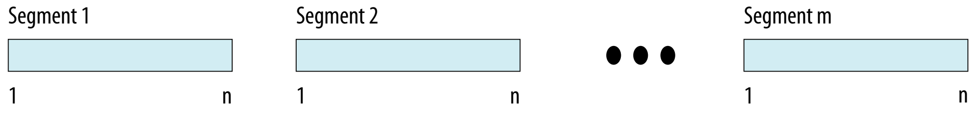
```
    increment c2;
    goto ST2;
end
```

Note: *m* x *n* must equal the sample depth to efficiently use the space in the sample buffer.

The next figure shows the segmented buffer that the trigger flow example describes.

Figure 81. Segmented Buffer Created with Storage Qualifier and State-Based Trigger

Total sample depth is fixed, where *m* x *n* must equal sample depth.



During runtime, you can modify the values m and n. Changing the m and n values in the trigger flow description adjust the segment boundaries without recompiling.

You can add states into the trigger flow description and selectively mask out specific states and enable other ones at runtime with status flags.

This example is like the previous example with an additional state inserted. You use this extra state to specify a different trigger condition that does not use the storage qualifier feature. You insert status flags into the conditional statements to control the execution of the trigger flow.

```
state ST1 :
    if (condition2  && f1)             // additional state for 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
```

2.7.3. Signal Tap Status Messages

The following table describes the text messages that might appear in the Signal Tap Status Indicator in the **Instance Manager** pane before, during, or after data acquisition. These messages allow you to monitor the state of the logic analyzer and identify the operation that the logic analyzer is performing.

Table 22. Messages in the Signal Tap Status Indicator

Message	Message Description
Not running	The Signal Tap logic analyzer is not running. This message appears when there is no connection to a device, or the device is not configured.
(Power-Up Trigger) Waiting for clock (1)	The Signal Tap logic analyzer is performing a Runtime or Power-Up Trigger acquisition and is waiting for the clock signal to transition.
Acquiring (Power-Up) pre-trigger data (1)	The trigger condition is not yet evaluated. If the acquisition mode is non-segmented buffer, and the storage qualifier type is continuous, the Signal Tap logic analyzer collects a full buffer of data.
Trigger In conditions met	Trigger In conditions are met. The Signal Tap logic analyzer is waiting for the first trigger condition to occur. This message only appears when a Trigger In condition exists.
Waiting for (Power-up) trigger (1)	The Signal Tap logic analyzer is waiting for the trigger event to occur.
Trigger level <x> met	Trigger condition *x* occurred. The Signal Tap logic analyzer is waiting for condition x + 1 to occur.
Acquiring (power-up) post-trigger data (1)	The entire trigger event occurred. The Signal Tap logic analyzer is acquiring the post-trigger data. You define the amount of post-trigger data to collect (between 12%, 50%, and 88%) when you select the non-segmented buffer acquisition mode.
Offload acquired (Power-Up) data (1)	The JTAG chain is transmitting data to the Intel Quartus Prime software.
Ready to acquire	The Signal Tap logic analyzer is waiting for you to initialize the analyzer.

1. This message can appear for both Runtime and Power-Up Trigger events. When referring to a Power-Up Trigger, the text in parentheses appears.

Note: In segmented acquisition mode, pre-trigger and post-trigger do not apply.

2.8. Step 6: Analyze Signal Tap Captured Data

The Signal Tap logic analyzer GUI allows you to examine the data that you capture manually or with a trigger. In the Data view, you can isolate the data of interest with the drag-to-zoom feature, enabled with a left-click. You can save the data for later analysis, or convert the data to other formats for sharing and further study.

- To simplify reading and interpreting the signal data you capture, set up mnemonic tables, either manually or with a plug-in.
- To speed up debugging, use the **Locate** feature in the **Signal Tap node** list to find the locations of problem nodes in other tools in the Intel Quartus Prime software.

The following topics describe viewing, saving, and exporting Signal Tap analysis captured data:

- Viewing Capture Data Using Segmented Buffers on page 91
- Viewing Data with Different Acquisition Modes on page 92
- Creating Mnemonics for Bit Patterns on page 93
- Locating a Node in the Design on page 94
- Saving Captured Signal Tap Data on page 95
- Exporting Captured Signal Tap Data on page 95
- Creating a Signal Tap List File on page 95

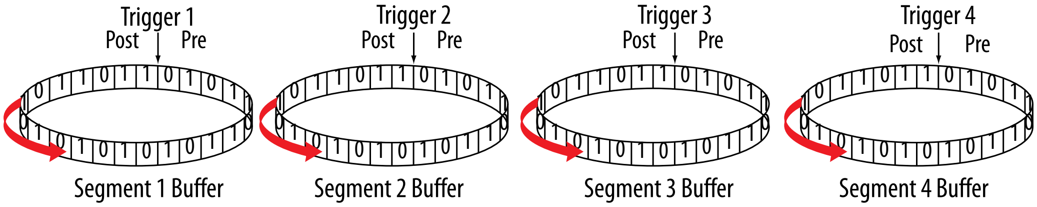
2.8.1. Viewing Capture Data Using Segmented Buffers

Segmented buffers allow you to capture recurring events or sequences of events that span over a long period.

Each acquisition segment acts as a non-segmented buffer, continuously capturing data after activation. When you run analyses with segmented buffers, the Signal Tap logic analyzer captures back-to-back data for each acquisition segment within the data buffer. You define the trigger flow, or the type and order in which the trigger conditions evaluate for each buffer, either in the Sequential trigger flow control or in the Custom State-based trigger flow control.

The following figure shows a segmented acquisition buffer with four segments represented as four separate non-segmented buffers.

Figure 82. Segmented Acquisition Buffer

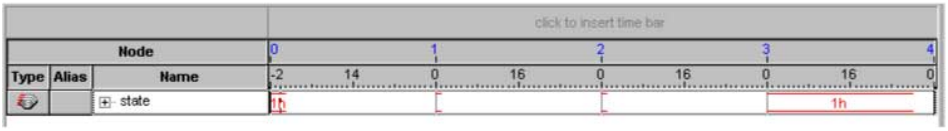


When the Signal Tap logic analyzer finishes an acquisition with a segment and advances to the next segment to start a new acquisition, the data capture that appears in the waveform viewer depends on when a trigger condition occurs. The figure illustrates the data capture method. The Trigger markers—Trigger 1, Trigger 2, Trigger 3 and Trigger 4—refer to the evaluation of the `segment_trigger` and `trigger` commands in the Custom State-based trigger flow. In sequential flows, the Trigger markers for segments 2 through 4 refer to the final trigger condition that you specify within the **Setup** tab.

If the Segment 1 Buffer is the active segment and Trigger 1 occurs, the Signal Tap logic analyzer starts evaluating Trigger 2 immediately. Data Acquisition for the Segment 2 buffer starts when either the Segment 1 Buffer finishes its post-fill count, or when Trigger 2 evaluates as TRUE, whichever condition occurs first. Thus, trigger conditions associated with the next buffer in the data capture sequence can preempt the post-fill count of the current active buffer. This allows the Signal Tap logic analyzer to accurately capture all the trigger conditions that occurred. Unused samples appear as a blank space in the waveform viewer.

Figure 83. Segmented Capture with Preemption of Acquisition Segments

The figure shows 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 (12% of the data is before the trigger condition and 88% of the data is after the trigger position). Because the last segment starts immediately with the trigger condition, the segment contains only post-trigger data. The three empty samples in the last segment are left over from the pre-trigger samples that the Signal Tap logic analyzer allocated to the buffer.

For the sequential trigger flow, the **Trigger Position** option applies to every segment in the buffer. A custom state-based trigger flow provides maximum flexibility defining the trigger position. By adjusting the trigger position specific to the debugging requirements, you can help maximize the use of the allocated buffer space.

Related Information

Segmented Buffer on page 44

2.8.2. Viewing Data with Different Acquisition Modes

Different acquisition modes capture different amounts of data immediately after running the Signal Tap logic analyzer and before any trigger conditions occur.

Non-Segmented Buffers in Continuous Mode

In configurations with non-segmented buffers running in continuous mode, the buffer must be full of sampled data before evaluating any trigger condition. Only after the buffer is full, the Signal Tap logic analyzer starts retrieving data through the JTAG connection and evaluates the trigger condition.

If you click the **Stop Analysis** button, Signal Tap prevents the buffer from dumping data during the first acquisition prior to a trigger condition.

Buffers with Storage Qualification

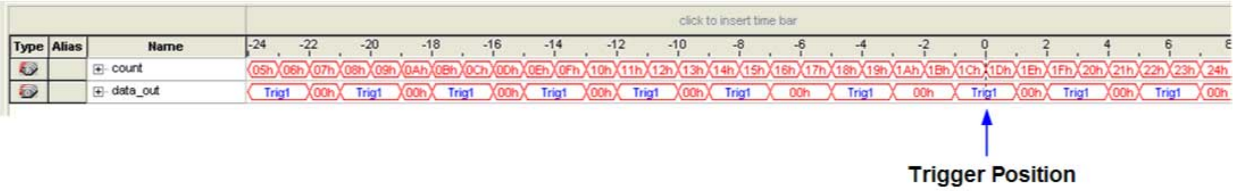
For buffers using a storage qualification mode, the Signal Tap logic analyzer immediately evaluates all trigger conditions while writing samples into the acquisition memory. This evaluation is especially important when using any storage qualification on the data set. The logic analyzer may miss a trigger condition if it waits to capture a full buffer's worth of data before evaluating any trigger conditions.

If a trigger activates before the specified amount of pre-trigger data has occurred, the Signal Tap logic analyzer 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 the target system, the trigger activates. However, the logic analyzer memory contains only post-trigger data, and not any pre-trigger data, because the trigger event has higher precedence than the capture of pre-trigger data.

2.8.2.1. Continuous Mode and a Storage Qualifier Examples

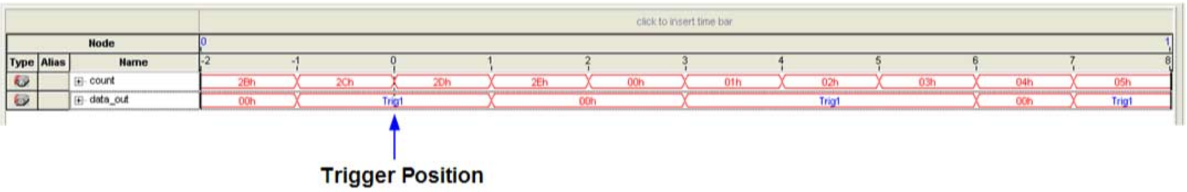
The following show the capture differences between a non-segmented buffer in continuous mode and a non-segmented buffer using a storage qualifier. The configuration of the logic analyzer waveforms is a base trigger condition, sample depth of 64 bits, and **Post trigger position**.

Figure 84. Signal Tap Logic Analyzer Continuous Data Capture


click to insert time bar
Type
Alias
Name
count
data_out
-24 -22 -20 -18 -16 -14 -12 -10 -8 -6 -4 -2 0 2 4 6 8
05h 06h 07h 08h 09h 0Ah 0Bh 0Ch 0Dh 0Eh 0Fh 10h 11h 12h 13h 14h 15h 16h 17h 18h 19h 1Ah 1Bh 1Ch 1Dh 1Eh 1Fh 20h 21h 22h 23h 24h
Trig1 00h Trig1 00h Trig1 00h Trig1 00h Trig1 00h Trig1 00h Trig1 00h Trig1 00h Trig1 00h Trig1 00h
Trigger Position

In the continuous data capture, Trig1 occurs several times in the data buffer before the Signal Tap logic analyzer trigger activates. The buffer must be full before the logic analyzer evaluates any trigger condition. After the trigger condition occurs, the logic analyzer continues acquisition for eight additional samples (12% of the buffer, as defined by the "post-trigger" position).

Figure 85. Signal Tap Logic Analyzer Conditional Data Capture



Note to figure:

1. Conditional capture, storage always enabled, post-fill count.
2. The Signal Tap logic analyzer captures a recurring pattern using a non-segmented buffer in conditional mode. The configuration of the logic analyzer is a basic trigger condition "Trig1" and sample depth of 64 bits. The **Trigger in** condition is **Don't care**, so the buffer captures all samples.

In conditional capture the logic analyzer triggers immediately. As in continuous capture, the logic analyzer completes the acquisition with eight samples, or 12% of 64, the sample capacity of the acquisition buffer.

2.8.3. Creating Mnemonics for Bit Patterns

A mnemonic table allows you to assign a meaningful name to a set of bit patterns, such as a bus. To create a mnemonic table:

1. Right-click the **Setup** or **Data** tab of a Signal Tap instance, and click **Mnemonic Table Setup**.
2. Create a mnemonic table by entering sets of bit patterns and specifying a label to represent each pattern.
3. Assign the table to a group of signals by right-clicking the group, clicking **Bus Display Format**, and selecting the mnemonic table.
4. 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.

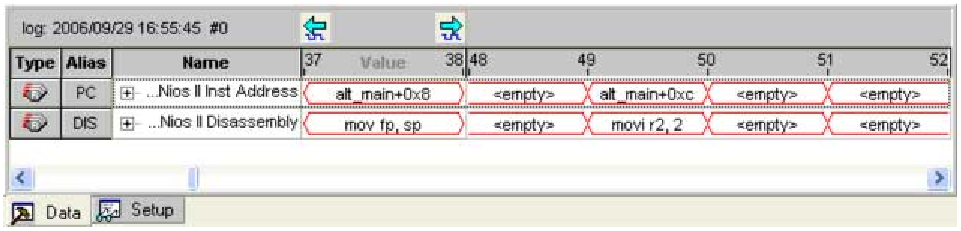
On the **Data** tab, if data captured matches a bit pattern contained in an assigned mnemonic table, the Signal Tap GUI replaces the signal group data with the appropriate label, simplifying the visual inspection of expected data patterns.

2.8.3.1. Adding Mnemonics with a Plug-In

When you use a plug-in to add signals to an `.stp`, mnemonic tables for the added signals are automatically created and assigned to the signals defined in the plug-in. To enable these mnemonic tables manually, right-click the name of the signal or signal group. On the **Bus Display Format** shortcut menu, 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 **Instruction Address** signal group at the trigger sample, along with the corresponding disassembled code in the **Disassembly** signal group, as shown in Figure 13–52. Captured data samples around the trigger are referenced as offset addresses from the trigger function name.

Figure 86. 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>	

2.8.4. Locating a Node in the Design

When you find the source of an error in your design using the Signal Tap logic analyzer, you can use the node locate feature to locate that signal in various Intel Quartus Prime design visualization tools, as well as in the design file. Locating the node allows you to visualize the source of the problem quickly and correct the issue. To locate a signal from the Signal Tap logic analyzer, right-click the signal in the `.stp`, and click **Locate in ➤ <tool name>**.

You can locate a signal from the node list with the following tools:

- Assignment Editor
- Pin Planner
- Timing Closure Floorplan
- Chip Planner
- Resource Property Editor
- Technology Map Viewer
- RTL Viewer
- Design File

2.8.5. Saving Captured Signal Tap Data

When you save a data capture, the Signal Tap logic analyzer stores this data in the active `.stp` file, and the **Data Log** adds the capture as a log entry under the current configuration.

When you set Signal Tap analysis to **Autorun Analysis**, which starts the Signal Tap logic analyzer in a repetitive acquisition mode, the logic analyzer creates a separate entry in the **Data Log** to store the data captured each time the trigger occurs. This preservation allows you to review the captured data for each trigger event.

The default name for a log derives from the time stamp when the logic analyzer acquires the data. As a best practice, rename the data log with a more meaningful name.

The organization of logs is hierarchical; the logic analyzer groups similar logs of captured data in trigger sets.

Related Information

Data Log Pane on page 99

2.8.6. Exporting Captured Signal Tap Data

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 the Signal Tap logic analyzer, click **File ➤ Export**, and then specify the **File Name**, **Export Format**, and **Clock Period**.

2.8.7. Creating a Signal Tap List File

You can generate a Signal Tap list file that contains all the data the logic analyzer captures for a trigger event, in text format.

The Signal Tap list file is especially useful when combined with a plug-in that includes instruction code disassembly. You can view the order of instruction code execution during the same time period of the trigger event.

To create a Signal Tap list file, click **File ➤ Create/Update ➤ Create Signal Tap List File**.

Each row of the list file corresponds to one captured sample in the buffer. Columns correspond to the value of each of the captured signals or signal groups for that sample. If you defined a mnemonic table for the captured data, a matching entry from the table replaces the numerical values in the list.

2.9. Other Signal Tap Debugging Flows

Refer to the following information about more advanced (non-standard) Signal Tap debugging flows and alternative methods.

2.9.1. Signal Tap and Simulator Integration

You can use Signal Tap signal and acquisition data directly in your supported simulator for enhanced visibility into internal signal states in a design hierarchy. The **Add Simulator Aware Nodes** command intelligently analyzes the circuit to determine the minimum set of nodes needed to tap to gain full visibility into the selected hierarchy's cone of logic.

Signal Tap can also transform the Signal Tap data into an RTL simulation testbench for any level of the design hierarchy. This simulation testbench allows you to export acquired Signal Tap hardware data directly into your RTL simulator and observe signal states beyond Signal Tap observability.

The following topics describe these Signal Tap and Simulator Integration features in detail:

- Adding Simulator-Aware Signal Tap Nodes on page 47
- Generating a Simulation Testbench from Signal Tap Data on page 96

Simulator Integration Beta Limitations

This version of the Signal Tap and simulator integration feature is a beta release. The following known limitations apply to this beta release:

- Supports only Verilog HDL simulation.
- Supports testbench generation only within the current project directory.

Related Information

Adding Signals to the Signal Tap Logic Analyzer on page 44

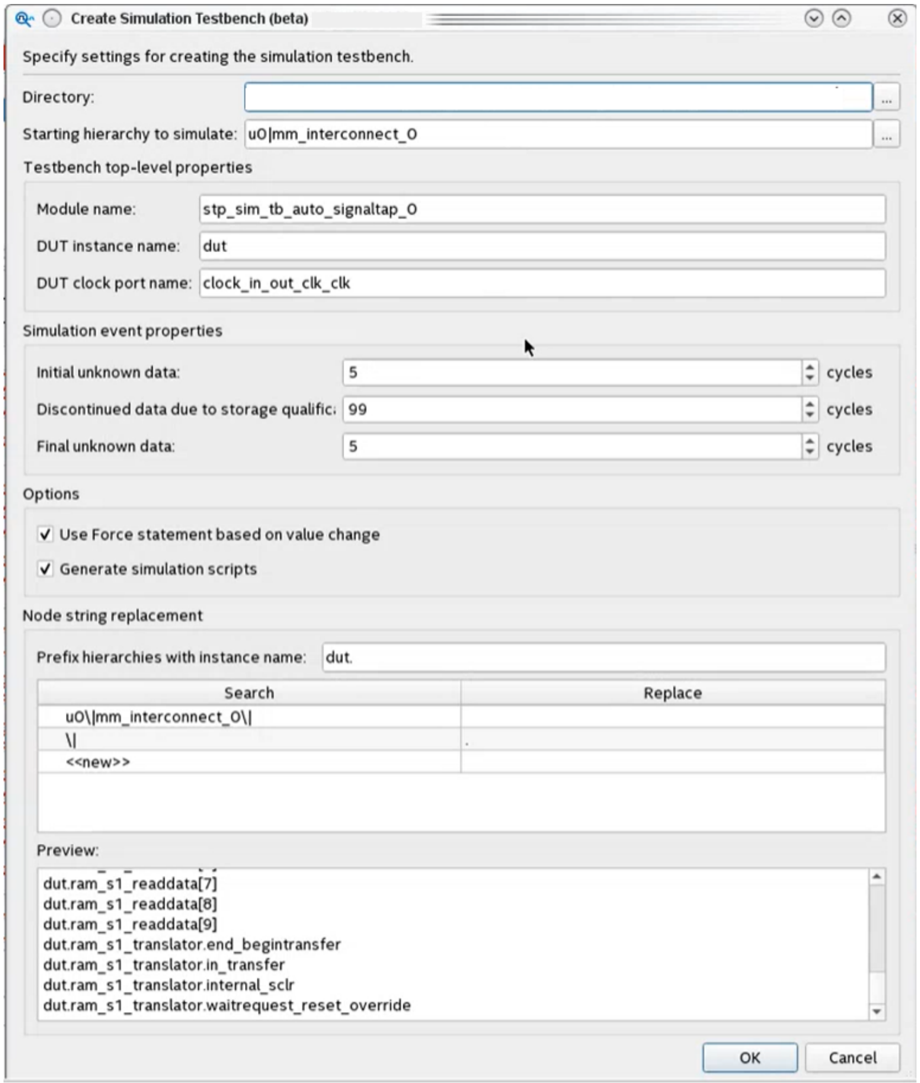
2.9.1.1. Generating a Simulation Testbench from Signal Tap Data

You can use Signal Tap to capture signal data about your running system, and then automatically generate an RTL simulation testbench directly from this capture data for use in your supported simulator.

To generate a simulation testbench from Signal Tap data, follow these steps:

1. Add simulator-aware Signal Tap nodes to the logic analyzer, as Adding Simulator-Aware Signal Tap Nodes on page 47 describes.
2. Run Signal Tap analysis, as Step 5: Run the Signal Tap Logic Analyzer on page 83 describes.

Figure 87. Create Simulation Testbench



Specify settings for creating the simulation testbench.

Directory:

Starting hierarchy to simulate: u0|mm_interconnect_0

Testbench top-level properties

Module name: stp_sim_tb_auto_signaltap_0

DUT instance name: dut

DUT clock port name: clock_in_out_clk_clk

Simulation event properties

Initial unknown data: 5 cycles

Discontinued data due to storage qualific: 99 cycles

Final unknown data: 5 cycles

Options

☑ Use Force statement based on value change

☑ Generate simulation scripts

Node string replacement

Prefix hierarchies with instance name: dut.

Search	Replace
u0|mm_interconnect_0|	
|	.
<<new>>	

Preview:

```
dut.ram_s1_readdata[7]
dut.ram_s1_readdata[8]
dut.ram_s1_readdata[9]
dut.ram_s1_translator.end_begintransfer
dut.ram_s1_translator.in_transfer
dut.ram_s1_translator.internal_sclr
dut.ram_s1_translator.waitrequest_reset_override
```

3. In the Signal Tap window, click **File ➤ Create Simulation Testbench**. Retain defaults and click **OK**. The testbench generates in a vendor-sepecific directory. Refer to Create Simulation Testbench Dialog Box Settings on page 98.
4. Source the generated simulator setup script in your supported simulator. For example:

```
source msim_setup.tcl
```

5. Use the commands in the setup script to compile and load the testbench into a supported simulator. For example, in the Questa or ModelSim simulators:

   ```
   ld_debug
   ```

   Note: Signal Tap uses a Verilog HDL `force` statement to inject the Signal Tap data into the simulator.

6. Add signals to the waveform and run the simulation in your simulator.
7. View the results of simulation in your simulator.

2.9.1.2. Create Simulation Testbench Dialog Box Settings

The following options are available for RTL simulation testbench generation from the Signal Tap **Create Simulation Testbench Dialog Box**. The default values derive from Signal Tap signal data.

Table 23. Create Simulation Testbench Dialog Box (Signal Tap Logic Analyzer)

Name	Description
Directory	Specifies the directory to generated save RTL simulation testbench files. *Note:* Signal Tap currently supports testbench generation only within the current project directory.
Starting hierarchy to simulate	Specifies the design hierarchy level to include in the simulation. The default location is a subdirectory of the project with the hierarchy name.
Testbench top level properties	Specifies the following testbench properties. By default, these values populate from the Signal Tap data: • **Module name**—specifies the name of the design module that you want to simulate, as specified in Signal Tap • **DUT instance name**—specifies the default instance name for the design under test (DUT) in your simulator. The default is **DUT**. This name appears in your simulator. • **DUT clock port name**—specifies the clock port name of the design under test (DUT) for simulation. Signal Tap automatically derives this value based on the **DUT instance name**.
Simulation event properties	Specifies the following testbench properties. By default, these values populate from the Signal Tap data: • **Initial unknown data**—specifies the number of clock cycles for which the data value is initially unknown at the start of simulation. • **Discontinued data due to storage qualification**—specifies the number of clock cycles for which the data is discontinued because of lack of storage. • **Final unknown data**—specifies the number of clock cycles for which the data is unknown initially at the end of simulation.
Options	The following options must be enabled for testbench generation: • **Use force statement based on value change**—specifies the number of clock cycles for which the data value is initially unknown at the start of simulation. *Note:* Signal Tap uses a Verilog HDL `force` statement to inject the Signal Tap data into the simulator. • **Generate simulation scripts**—specifies that simulation scripts generate in vendor specific subdirectories during testbench generation. Source these scripts in your simulator to setup simulation.
Node string replacement	Specifies options for nomenclature and syntax within the generated testbench:

continued...

Name	Description
	• **Prefix hierarchies with instance name**—specifies the instance name that prepends to hierarchy names in the testbench. In general, the derived default value is suitable. • **Search|Replace**—specifies the search and replace strings for **Node string replacement**.
Preview	Displays the result of the **Node string replacement** settings within the testbench.

2.9.2. Managing Multiple Signal Tap Configurations

You can debug different blocks in your design by grouping related monitoring signals. Similarly, you can use a group of signals to define multiple trigger conditions. Each combination of signals, capture settings, and trigger conditions determines a debug configuration, and one configuration can have zero or more associated data logs.

You can save each debug configuration as a different `.stp` file. Alternatively, you can embed multiple configurations within the same `.stp` file, and use the **Data Log** to view and manage each debug configuration.

Note: Each `.stp` pertains to a specific programming (`.sof`) file. To function correctly, the settings in the `.stp` file you use at runtime must match the Signal Tap specifications in the `.sof` file that you use to program the device.

Related Information

Ensure Compatibility Between .stp and .sof Files on page 83

2.9.2.1. Data Log Pane

The **Data Log** pane displays all Signal Tap configurations and data capture results that a single `.stp` file stores.

- To save the current configuration or capture in the **Data Log** of the current `.stp` file, click **Edit ➤ Save to Data Log**.
- To automatically generate a log entry after every data capture, click **Edit ➤ Enable Data Log**. Alternatively, enable the box at the top of the **Data Log** pane.

The **Data Log** displays its contents in a tree hierarchy. The active items display a different icon.

Table 24. Data Log Items

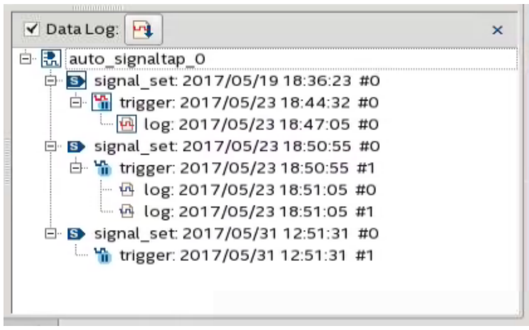
Item	Icon		Contains one or more	Comments
	Unselected	Selected		
Instance			Signal Set	The top-level for a particular Signal Tap instance.
Signal Set	S	S	Trigger	The Signal Set changes whenever you add a new signal to a Signal Tap instance. After a change in the Signal Set, you need to recompile.
Trigger			Capture Log	A trigger changes when you change any trigger condition. Some of these changes do not require recompilation.
Capture Log				Contains captured sample data for this particular trigger configuration, for the particular signal set for this particular Signal Tap instance. There can be multiple capture logs for a particular setup if you run the logic analyzer multiple times, as Figure 88 on page 100 shows.

The name on each entry displays the wall-clock time when the Signal Tap logic analyzer triggers, and the time elapsed from start acquisition to trigger activation. You can rename entries.

To switch between configurations, double-click an entry in the **Data Log**. As a result, the **Setup** and **Data** tabs update to display the active signal list, trigger conditions, or specified captured data.

Figure 88. Simple Data Log

In this example, the **Data Log** displays one instance with three signal set configurations, two trigger condition setups, and three different captured data sets.



```
☑ Data Log:
auto_signaltap_0
  signal_set: 2017/05/19 18:36:23 #0
    trigger: 2017/05/23 18:44:32 #0
      log: 2017/05/23 18:47:05 #0
  signal_set: 2017/05/23 18:50:55 #0
    trigger: 2017/05/23 18:50:55 #1
      log: 2017/05/23 18:51:05 #0
      log: 2017/05/23 18:51:05 #1
  signal_set: 2017/05/31 12:51:31 #0
    trigger: 2017/05/31 12:51:31 #1
```

2.9.2.2. SOF Manager

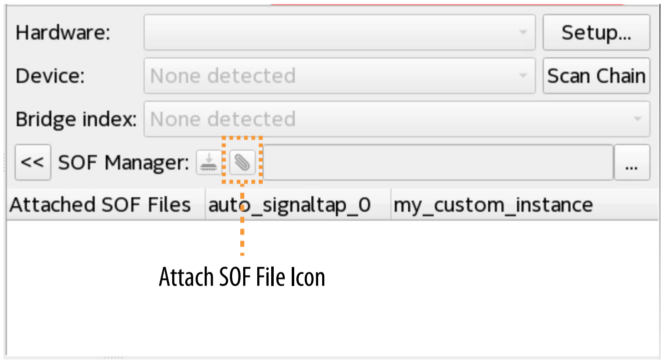
The SOF Manager is in the **JTAG Chain Configuration** pane.

With the SOF Manager you can attach multiple .sof files to a single .stp file. This attachment allows you to move the .stp file to a different location, either on the same computer or across a network, without including the attached .sof separately.

The SOF Manager also ensures compatibility between any attached .sof files and the current .stp file settings automatically, asEnsure Compatibility Between .stp and .sof Files on page 83 describes.

To attach a new .sof in the .stp file, click the **Attach SOF File** icon .

Figure 89. SOF Manager



Hardware:

Device: None detected

Bridge index: None detected

SOF Manager:

Attached SOF Files | auto_signaltap_0 | my_custom_instance

As you switch between configurations in the **Data Log**, you can extract the `.sof` that is compatible with that configuration.

To download the new `.sof` to the FPGA, click the **Program Device** icon in the SOF Manager, after ensuring that the configuration of your `.stp` is compatible with the design to program into the target device.

Related Information

Data Log Pane on page 99

2.9.3. Debugging Partial Reconfiguration Designs with Signal Tap

You can debug a Partial Reconfiguration (PR) design with the Signal Tap logic analyzer. The Signal Tap logic analyzer supports data acquisition in the static and PR regions. You can debug multiple personas present in a PR region and multiple PR regions.

For examples on debugging PR designs targeting specific devices, refer to *AN 841: Signal Tap Tutorial for Intel Stratix® 10 Partial Reconfiguration Design* or *AN 845: Signal Tap Tutorial for Intel Arria® 10 Partial Reconfiguration Design*.

Related Information

- AN 841: Signal Tap Tutorial for Intel Stratix 10 Partial Reconfiguration Design
- AN 845: Signal Tap Tutorial for Intel Arria 10 Partial Reconfiguration Design

2.9.3.1. Signal Tap Guidelines for PR Designs

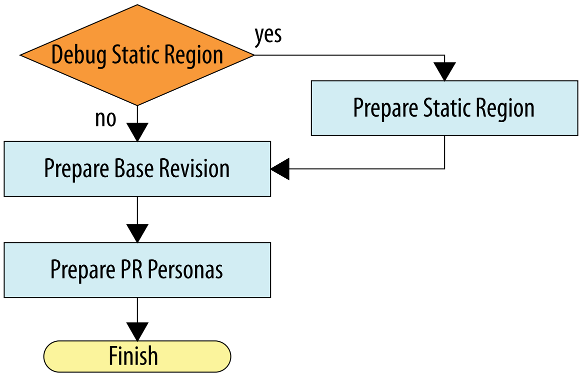
Follow these guidelines to obtain the best results when debugging PR designs with the Signal Tap logic analyzer:

- Include one `.stp` file per project revision.
- Tap pre-synthesis nodes only. In the Node Finder, filter by **Signal Tap: pre-synthesis**.
- Do not tap nodes in the default persona (the personas you use in the base revision compile). Create a new PR implementation revision that instantiates the default persona, and tap nodes in the new revision.

- Store all the tapped nodes from a PR persona in one `.stp` file, to enable debugging the entire persona using only one Signal Tap window.
- Do not tap across PR regions, or from a static region to a PR region in the same `.stp` file.
- Each Signal Tap window opens only one `.stp` file. Therefore, to debug more than one partition simultaneously, you must use stand-alone Signal Tap from the command-line.

2.9.3.2. PR Design Setup for Signal Tap Debug

Figure 90. Setting Up PR Design for Debug with Signal Tap



To debug a PR design, you must instantiate SLD JTAG bridges when generating the base revision, and then define debug components for all PR personas. Optionally, you can specify signals to tap in the static region. After configuring all the PR personas in the design, you can continue the PR design flow.

Related Information

- Debug Fabric for Partial Reconfiguration Designs on page 19
- Partial Reconfiguration Design Flow, Intel Quartus Prime Pro Edition User Guide: Partial Reconfiguration

2.9.3.2.1. Preparing the Static Region for Signal Tap Debugging

To debug the static region in your PR design:

1. Tap nodes in the static region exclusively.
2. Save the `.stp` file with a name that identifies the file with the static region.
3. Enable Signal Tap in your project, and include the `.stp` file in the base revision.

Note: Do not tap signals in the default PR personas.

2.9.3.2.2. Preparing the Base Revision for Signal Tap Debugging

In the base revision, for each PR region that you want to debug in the design:

1. Instantiate the SLD JTAG Bridge Agent Intel FPGA IP in the static region.
2. Instantiate the SLD JTAG Bridge Host Intel FPGA IP in the PR region of the default persona.

You can use the IP Catalog or Platform Designer to instantiate SLD JTAG Bridge components.

Related Information

- Instantiating the SLD JTAG Bridge Agent on page 16
- Instantiating the SLD JTAG Bridge Host on page 17

2.9.3.2.3. Preparing PR Personas for Signal Tap Debugging

Before you create revisions for personas in your design, you must instantiate debug IP components and tap signals.

For each PR persona that you want to debug:

1. Instantiate the SLD JTAG Bridge Host Intel FPGA IP in the PR persona.
2. Tap pre-synthesis nodes in the PR persona only.
3. Save in a new `.stp` file with a name that identifies the persona.
4. Use the new `.stp` file in the implementation revision.

If you do not want to debug a particular persona, drive the `tdo` output signal to 0.

2.9.3.3. Performing Data Acquisition in a PR design

After generating the `.sof` and `.rbf` files for the revisions you want to debug, you are ready to program your device and debug with the Signal Tap logic analyzer.

To perform data acquisition:

1. Program the base image into your device.
2. Partially reconfigure the device with the persona that you want to debug.
3. Open the Signal Tap logic analyzer by clicking **Tools ➤ Signal Tap logic analyzer** in the Intel Quartus Prime software.

   The logic analyzer opens and loads the `.stp` file set in the current active revision.
4. To debug other regions in your design, open new Signal Tap windows by opening the other region's `.stp` file from the Intel Quartus Prime main window.

   Alternatively, use the command-line:

   ```
   quartus_stpw <stp_file_other_region.stp>
   ```
5. Debug your design with Signal Tap.

To debug another revision, you must partially reconfigure your design with the corresponding `.rbf` file.

2.9.4. Debugging Block-Based Designs with Signal Tap

The Intel Quartus Prime Pro Edition software supports verification of block-based design flows with the Signal Tap logic analyzer.

Verifying a block-based design requires planning to ensure visibility of logic inside partitions and communication with the Signal Tap logic analyzer. The preparation steps depend on whether you are reusing a core partition or a root partition.

For information about designing with reusable blocks, refer to the *Intel Quartus Prime Pro Edition User Guide: Block-Based Design*. For step-by-step block-based design debugging instructions, refer to *AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria 10 FPGA Development Board*.

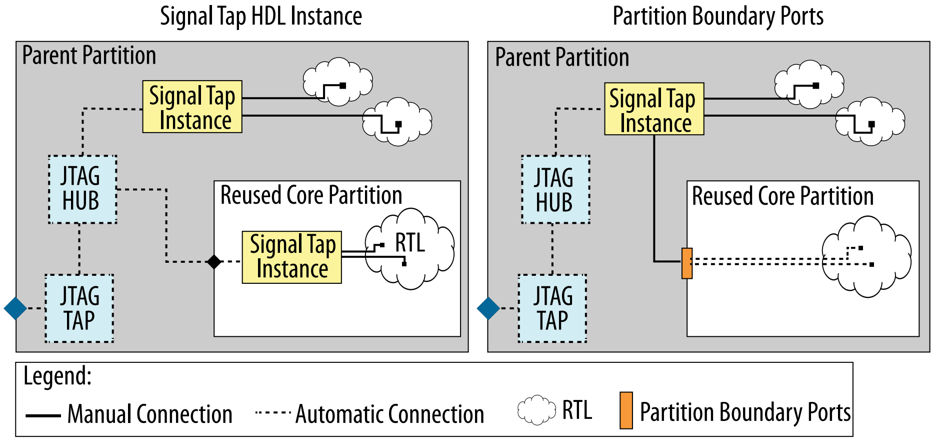
Related Information

Intel Quartus Prime Pro Edition User Guide: Block-Based Design

2.9.4.1. Signal Tap Debugging with a Core Partition

To perform Signal Tap debugging in a core design partition that you reuse from another project, you identify the signals of interest, and then make those signals visible to a Signal Tap logic analyzer instance. The Intel Quartus Prime software supports two methods to make the reused core partition signals visible for Signal Tap monitoring: by creating partition boundary ports, or by Signal Tap HDL instantiation.

Figure 91. Debug Setup with Reused Core Partition



2.9.4.1.1. Partition Boundary Ports Method

Partition boundary ports expose core partition nodes to the top-level partition. Boundary ports simplify the management of hierarchical blocks by tunneling through layers of logic without making RTL changes. The partition boundary ports method includes these high-level steps:

1. In the project that exports the partition, define boundary ports for all potential Signal Tap nodes in the core partition. Define partition boundary ports with the **Create Partition Boundary Ports** assignment in the Assignment Editor. When you assign a bus, the assignment applies to the root name of the debug port, with each bit enumerated.
2. In the project that exports the partition, create a black box file that includes the partition boundary ports, to allows tapping these ports as pre-synthesis or post-fit nodes in another project.
3. In the project that reuses the partition, run Analysis & Synthesis on the reused partition. All valid ports with the **Create Partition Boundary Ports** become visible in the project. After synthesis you can verify the partition boundary ports in the Create Partition Boundary Ports report in the **In-System Debugging** folder under **Synthesis** reports.
4. Tap the partition boundary ports to connect to a Signal Tap instance in the top-level partition. You can also tap logic from the top-level partition to this Signal Tap instance. When using this method, the project requires only one Signal Tap instance to debug both the top-level and the reused core partition.

The following procedures explain these steps in more detail.

2.9.4.1.2. Debug a Core Partition through Partition Boundary Ports

To use Signal Tap to debug a design that includes a core partition exported with partition boundary ports from another project, follow these steps:

1. Add to your project the black-box file that you create in Export a Core Partition with Partition Boundary Ports on page 105.
2. To run synthesis, double-click **Analysis & Synthesis** on the Compilation Dashboard.
3. Define a Signal Tap instance with the Signal Tap GUI, or by instantiating a Signal Tap HDL instance in the top level root partition, as Step 1: Add the Signal Tap Logic Analyzer to the Project on page 33 describes.
4. Connect the partition boundary ports of the reused core partition to the HDL instance, or add post-synthesis or post-fit nodes to the **Signal Configuration** tab in the Signal Tap logic analyzer GUI.
5. To create a design partition, click **Assignments ➤ Design Partitions Window**. Define a partition and assign the exported partition `.qdb` file as the **Partition Database File** option.
6. Compile the design, including all partitions and the Signal Tap instance.
7. Program the Intel FPGA device with the design and Signal Tap instances.
8. Perform data acquisition with the Signal Tap logic analyzer GUI.

2.9.4.1.3. Export a Core Partition with Partition Boundary Ports

To export a core partition with partition boundary ports for reuse and Signal Tap debugging in another project, follow these steps:

1. To run synthesis, double-click **Analysis & Synthesis** on the Compilation Dashboard.
2. Define a design partition for reuse that contains only core logic. Click **Assignments ➤ Design Partitions Window** to define the partition.
3. To create partition boundary ports for the core partition, specify the **Create Partition Boundary Ports** assignment in the Assignment Editor for partition ports.
4. Click **Project ➤ Export Design Partition**. By default, the `.qdb` file you export includes any Signal Tap HDL instances for the partition.
5. Compile the design and Signal Tap instance.
6. Create a black box file that defines only the port and module or entity definitions, without any logic.
7. Manually copy the exported partition `.qdb` file and any black box file to the other project.

Optionally, you can verify signals in the root and core partitions in the Developer project with the Signal Tap logic analyzer.

2.9.4.1.4. Signal Tap HDL Instance Method

To use the Signal Tap HDL instance method, you first create a Signal Tap HDL instance in the reusable core partition, and then connect the signals of interest to that instance. The Compiler ensures top-level visibility of Signal Tap instances inside partitions. Since the root partition and the core partition have separated HDL instances, the Signal Tap files are also separate.

When you reuse the partition in another project, you must generate one Signal Tap file in the target project for each HDL instance present in the reused partition.

Debug a Core Partition Exported with Signal Tap HDL Instances

To use Signal Tap to debug a design that includes a core partition exported with Signal Tap HDL instances, follow these steps:

1. Add to your project the black-box file that you create in Export a Core Partition with Signal Tap HDL Instances on page 107.
2. To create a design partition, click **Assignments ➤ Design Partitions Window**. Define a partition and assign the exported partition `.qdb` file as the **Partition Database File** option.
3. Create a Signal Tap file for the top-level partition as Step 1: Add the Signal Tap Logic Analyzer to the Project on page 33 describes.
4. Compile the design and Signal Tap instances.
5. Generate a Signal Tap file for the reused Core Partition with the **File ➤ Create/ Update ➤ Create Signal Tap File from Design Instance** command.
6. Program the Intel FPGA device with the design and Signal Tap instances.
7. Perform hardware verification of top-level partition with the Signal Tap instance defined in Step 3.
8. Perform hardware verification of the Reused Core Partition with the Signal Tap instance defined in Step 5.

2.9.4.1.5. Export a Core Partition with Signal Tap HDL Instances

To export a core partition with Signal Tap HDL instances for reuse and eventual Signal Tap debugging in another project, follow these steps:

1. To run synthesis, double-click **Analysis & Synthesis** on the Compilation Dashboard.
2. Define a design partition for reuse that contains only core logic. Click **Assignments ➤ Design Partitions Window** to define the partition.
3. Add a Signal Tap HDL instance to the core partition, connecting it to nodes of interest.
4. Click **Project ➤ Export Design Partition**. By default, the `.qdb` file you export includes any Signal Tap HDL instances for the partition.
5. Create a black box file that defines only the port and module or entity definitions, without any logic.
6. Manually copy the exported partition `.qdb` file and any black box file to the other project.

2.9.4.1.6. Debug a Core Partition Exported with Signal Tap HDL Instances

To use Signal Tap to debug a design that includes a core partition exported with Signal Tap HDL instances, follow these steps:

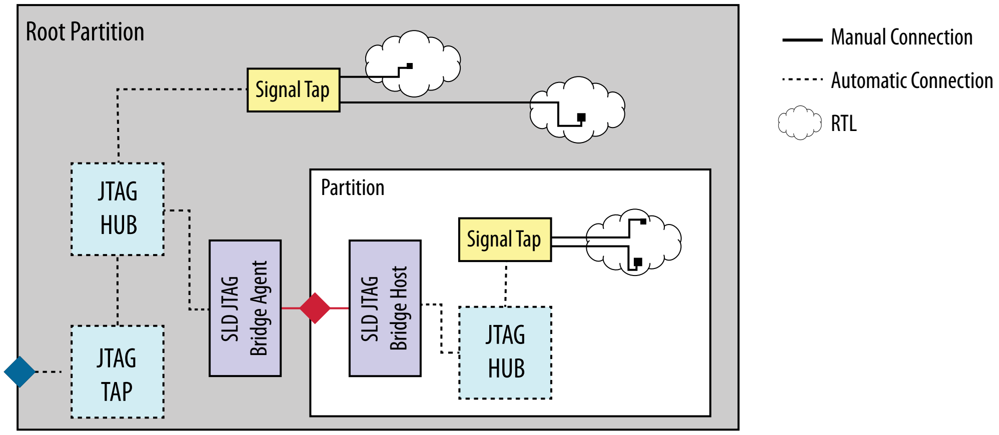
1. Add to your project the black-box file that you create in Export a Core Partition with Signal Tap HDL Instances on page 107.
2. To create a design partition, click **Assignments ➤ Design Partitions Window**. Define a partition and assign the exported partition `.qdb` file as the **Partition Database File** option.
3. Create a Signal Tap file for the top-level partition as Step 1: Add the Signal Tap Logic Analyzer to the Project on page 33 describes.
4. Compile the design and Signal Tap instances.
5. Generate a Signal Tap file for the reused Core Partition with the **File ➤ Create/ Update ➤ Create Signal Tap File from Design Instance** command.
6. Program the Intel FPGA device with the design and Signal Tap instances.
7. Perform hardware verification of top-level partition with the Signal Tap instance defined in Step 3.
8. Perform hardware verification of the Reused Core Partition with the Signal Tap instance defined in Step 5.

2.9.4.2. Signal Tap Debugging with a Root Partition

In a project that reuses a root partition, you enable debugging of the root partition and the core partition independently, with separate Signal Tap instances in each partition. In the project that exports the partition, you add the Signal Tap instance to the root partition. Additionally, you extend the debug fabric into the reserved core partition with a debug bridge. This bridge allows subsequent instantiation of Signal Tap when reusing the partition in another project.

You implement the debug bridge with the SLD JTAG Bridge Agent Intel FPGA IP and SLD JTAG Bridge Host Intel FPGA IP pair for each reserved core boundary in the design. You instantiate the SLD JTAG Bridge Agent IP in the root partition, and the SLD JTAG Bridge Host IP in the core partition.

Figure 92. Debug Setup with Reused Root Partition



For details about the debug bridge, refer to the *SLD JTAG Bridge* in the *System Debugging Tools Overview* chapter.

Related Information

SLD JTAG Bridge on page 14

2.9.4.2.1. Export the Root Partition with SLD JTAG Bridge

To export a reusable root partition with SLD JTAG Bridge that allows debugging of core partitions in another project, follow these steps.

1. Create a reserved core partition and define a Logic Lock region.
2. Generate and instantiate SLD JTAG Bridge Agent in the root partition.

   The combination of agent and host allows debugging the reserved core partition in Consumer projects.
3. Generate and instantiate the SLD JTAG Bridge Host in the reserved core partition.
4. Add a Signal Tap instance to the root partition, as Step 1: Add the Signal Tap Logic Analyzer to the Project on page 33 describes.
5. In the Signal Tap instance, specify the signals for monitoring. This action allows debugging the root partition in the Developer and Consumer projects.
6. Compile the design and Signal Tap instance.
7. Click **Project ➤ Export Design Partition**. By default, the .qdb file you export includes any Signal Tap HDL instances for the partition.
8. Manually copy files to the project that reuses the root partition:
   - In designs targeting the Intel Arria 10 device family, copy .qdb and .sdc files.
   - In designs targeting the Intel Stratix 10 device family copy the .qdb file.

In designs with multiple child partitions, you must provide the hierarchy path and the associated index of the JTAG Bridge Instance Agents in the design to the Consumer.

2.9.4.2.2. Debugging an Exported Root Partition and Core Partition Simultaneously using the SLD JTAG Bridge

When you reuse an exported root partition in another project, the exported `.qdb` includes the Signal Tap connection to signals in the root partition, and the SLD JTAG Bridge Agent IP, which allows debugging logic in the core partition.

To perform Signal Tap debugging in a project that includes a reused root partition:

1. Add the exported `.qdb` (and `.sdc`) files to the project that reuses them.
2. From the IP Catalog, parameterize and instantiate the SLD JTAG Bridge Host Intel FPGA IP in the core partition.
3. Run the Analysis & Synthesis stage of the Compiler.
4. Create a Signal Tap instance in the core partition, as Step 1: Add the Signal Tap Logic Analyzer to the Project on page 33 describes.
5. In the Signal Tap instance, specify post-synthesis signals for monitoring.

   Note: You can only tap signals in the core partition.
6. Compile the design and Signal Tap instance.
7. Generate a Signal Tap file for the reused root partition with the `quartus_stp` command.
8. Program the device.
9. Perform hardware verification of the reserved core partition with the Signal Tap instance defined in Step 3.
10. Perform hardware verification of the reused root partition with the Signal Tap instance defined in Step 7.

2.9.4.3. Compiler Snapshots and Signal Tap Debugging

When you reuse a design partition exported from another project, the design partition preserves the results of a specific snapshot of the compilation. Whenever possible, it is easiest to specify the signals for monitoring in the original project that exports the partition.

Adding new signals to a Signal Tap instance in a reused partition requires the Fitter to connect and route these signals. This is only possible when:

- The reused partition contains the Synthesis snapshot—reused partitions that contain the Placed or Final snapshot do not allow adding more signals to the Signal Tap instance for monitoring, because you cannot create additional boundary ports.
- The signal that you want to tap is a post-fit signal—adding pre-synthesis Signal Tap signals is not possible, because that requires resynthesis of the partition.

Related Information

Signals Unavailable for Signal Tap Debugging on page 50

2.9.4.3.1. Add Post-Fit Nodes when Reusing a Partition Containing a Synthesis Snapshot

You can add post-fit nodes for Signal Tap debug when reusing a design partition containing the synthesis snapshot exported from another project.

To add post-fit nodes to Signal Tap for monitoring:

1. Open the project that reuses the partition, and then compile the reused partition through the Fitter stage.
2. Add a Signal Tap instance to the project that reuses the partition, as Step 1: Add the Signal Tap Logic Analyzer to the Project on page 33 describes.
3. In the Signal Tap GUI, add the post-fit Signal Tap nodes to the **Signal Configuration** tab.
4. Recompile the design from the Place stage by clicking **Processing ➤ Start ➤ Start Fitter (Place)**.

   The Fitter attaches the Signal Tap nodes to the existing synthesized nodes.

2.9.5. Debugging Devices that use Configuration Bitstream Security

Some Intel FPGA device families support bitstream decryption during configuration using an on-device AES decryption engine. You can still use the Signal Tap logic analyzer to analyze functional data within the FPGA with such devices. However, JTAG configuration is not possible after programming the security key into the device.

Use an unencrypted bitstream during the prototype and debugging phases of the design, to allow programming file generation and reconfiguration of the device over the JTAG connection while debugging.

If you must use the Signal Tap logic analyzer with an encrypted bitstream, first configure the device with an encrypted configuration file using Passive Serial (PS), Fast Passive Parallel (FPP), or Active Serial (AS) configuration modes. The design must contain at least one instance of the Signal Tap logic analyzer. After configuring the FPGA with a Signal Tap instance and the design, you can open the Signal Tap logic analyzer GUI and scan the chain to acquire data with the JTAG connection.

Related Information

Intel Quartus Prime Pro Edition User Guide: Programmer

2.9.6. Signal Tap Data Capture with the MATLAB MEX Function

When you use MATLAB for DSP design, you can acquire data from the Signal Tap logic analyzer directly into a matrix in the MATLAB environment. To use this method, you call the MATLAB MEX function, `alt_signaltap_run`, that the Intel Quartus Prime software includes. If you use the MATLAB MEX function in a loop, you can perform as many acquisitions in the same amount of time as when using Signal Tap in the Intel Quartus Prime software environment.

Note: The MATLAB MEX function for Signal Tap is available in the Windows* version and Linux version of the Intel Quartus Prime software. This function is compatible with MATLAB Release 14 Original Release Version 7 and later.

To set up the Intel Quartus Prime software and the MATLAB environment to perform Signal Tap acquisitions:

1. In the Intel Quartus Prime software, create an `.stp` file.
2. In the node list in the **Data** tab of the Signal Tap logic analyzer Editor, organize the signals and groups of signals into the order in which you want them to appear in the MATLAB matrix.

   Each column of the imported matrix represents a single Signal Tap acquisition sample, while each row represents a signal or group of signals in the order you defined in the **Data** tab.

   Note: Signal groups that the Signal Tap logic analyzer acquires and transfers into the MATLAB MEX function have a width limit of 32 signals. To use the MATLAB MEX function with a bus or signal group that contains more than 32 signals, split the group into smaller groups that do not exceed the limit.

3. Save the `.stp` file and compile your design. Program your device and run the Signal Tap logic analyzer to ensure your trigger conditions and signal acquisition work correctly.
4. In the MATLAB environment, add the Intel Quartus Prime binary directory to your path with the following command:

   ```
   addpath <Quartus install directory>\win
   ```

   You can view the help file for the MEX function by entering the following command in MATLAB without any operators:

   `alt_signaltap_run`

5. Use the MATLAB MEX function to open the JTAG connection to the device and run the Signal Tap logic analyzer to acquire data. When you finish acquiring data, close the JTAG connection.

   To open the JTAG connection and begin acquiring captured data directly into a MATLAB matrix called `stp`, use the following command:

   ```
   stp = alt_signaltap_run \
   ('<stp filename>'[,('signed'|'unsigned')[,'<instance names>'[, \
   '<signalset name>'[,'<trigger name>']]]]);
   ```

   When capturing data, you must assign a filename, for example, *<stp filename>* as a requirement of the MATLAB MEX function. The following table describes other MATLAB MEX function options:

Table 25. Signal Tap MATLAB MEX Function Options

Option	Usage	Description
`signed` `unsigned`	`'signed'` `'unsigned'`	The **signed** option turns signal group data into 32-bit two's-complement signed integers. The MSB of the group as defined in the Signal Tap **Data** tab is the sign bit. The **unsigned** option keeps the data as an unsigned integer. The default is **signed**.
<instance name>	`'auto_signaltap_0'`	Specify a Signal Tap instance if more than one instance is defined. The default is the first instance in the `.stp`, `auto_signaltap_0`.
<signal set name> *<trigger name>*	`'my_signalset'` `'my_trigger'`	Specify the signal set and trigger from the Signal Tap data log if multiple configurations are present in the `.stp`. The default is the active signal set and trigger in the file.

During data acquisition, you can enable or disable verbose mode to see the status of the logic analyzer. To enable or disable verbose mode, use the following commands:

```
alt_signaltap_run('VERBOSE_ON');-alt_signaltap_run('VERBOSE_OFF');
```

When you finish acquiring data, close the JTAG connection with the following command:

```
alt_signaltap_run('END_CONNECTION');
```

For more information about the use of MATLAB MEX functions in MATLAB, refer to the MATLAB Help.

2.10. Signal Tap Logic Analyzer Design Examples

Application Note 845: Signal Tap Tutorial for Intel Arria 10 Partial Reconfiguration Design includes a design example that demonstrates Signal Tap debugging with a partial reconfiguration design. The design example has one 32-bit counter. At the board level, the design connects the clock to a 50MHz source, and connects the output to four LEDs on the FPGA. Selecting the output from the counter bits in a specific sequence causes the LEDs to blink at a specific frequency example demonstrates initiating a DMA transfer. The tutorial demonstrates how to tap signals in a PR design by extending the debug fabric to the PR regions when creating the base revision, and then defining debug components in the implementation revisions.

Application Note 446: Debugging Nios II Systems with the Signal Tap Logic Analyzer includes a design example with a Nios II processor, a direct memory access (DMA) controller, on-chip memory, and an interface to external SDRAM memory. After you press a button, the processor initiates a DMA transfer, which you analyze using the Signal Tap logic analyzer. In this example, the Nios II processor executes a simple C program from on-chip memory and waits for you to press a button.

Related Information

- AN 845: Signal Tap Tutorial for Intel Arria 10 Partial Reconfiguration Design
- AN 446: Debugging Nios II Systems with the Signal Tap Logic Analyzer

2.11. Custom State-Based Triggering Flow Examples

The custom state-based triggering flow in the Signal Tap logic analyzer GUI can organize multiple triggering conditions for precise control over the acquisition buffer. The following examples demonstrate defining a custom triggering flow. You can easily copy the examples directly into the state machine description box by specifying the **All states in one window** option.

Related Information

On-chip Debugging Design Examples website

2.11.1. Trigger Example 1: 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.

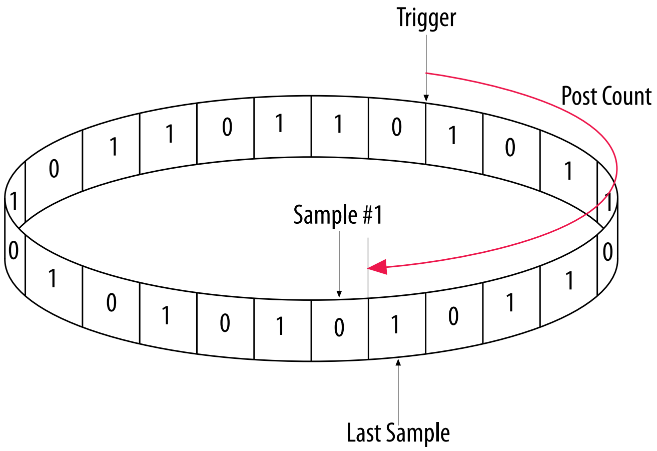
The following example shows how to apply a trigger position to all segments in the acquisition buffer. The example describes a triggering flow for an acquisition buffer split into four segments. If each acquisition segment is 64 samples in depth, the trigger position for each buffer is at sample #34. The acquisition stops after all segments are filled once.

```
if (c1 == 3 && condition1)
    trigger 30;
else if ( condition1 )
begin
    segment_trigger 30;
    increment c1;
end
```

Each segment acts as a non-segmented buffer that continuously updates the memory contents with the signal values.

The **Data** tab displays the last acquisition before stopping the buffer as the last sample number in the affected segment. The trigger position in the affected segment is then defined by *N* `– post count fill`, where *N* is the number of samples per segment.

Figure 93. Specifying a Custom Trigger Position



2.11.2. Trigger Example 2: Trigger When triggercond1 Occurs Ten Times between triggercond2 and triggercond3

You can use a custom trigger flow to count a sequence of events before triggering the acquisition buffer, as the following example shows. This example uses three basic triggering conditions configured in the Signal Tap **Setup** tab.

This example triggers the acquisition buffer when `condition1` occurs after `condition3` and occurs ten times prior to `condition3`. If `condition3` occurs prior to ten repetitions of `condition1`, the state machine transitions to a permanent wait state.

```
state ST1:
if ( condition2  )
begin
   reset c1;
   goto ST2;
end
```

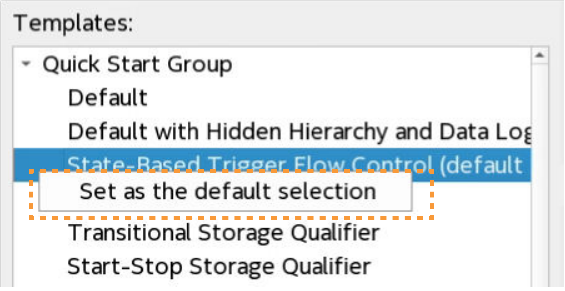
```
State ST2 :
if ( condition1 )
    increment c1;
else if (condition3 && c1 < 10)
    goto ST3;
else if ( condition3 && c1 >= 10)
    trigger;
ST3:
goto ST3;
```

2.12. Signal Tap File Templates

Signal Tap file templates provide preset settings for various trigger conditions, interfaces, and state-based triggering flows. The following Signal Tap file templates are available whenever you open a new Signal Tap session or create a new .stp file.

Right-click any template in the **New File from Template** dialog box, and then click **Set as the default selection** to always open new .stp files in that template by default.

Figure 94. Settings Default Signal Tap File Template



Templates:

- Quick Start Group
  - Default
  - Default with Hidden Hierarchy and Data Log
  - State-Based Trigger Flow Control (default
  - Set as the default selection
  - Transitional Storage Qualifier
  - Start-Stop Storage Qualifier

Note: Refer to the **New File from Template** dialog box for complete descriptions of all templates.

Table 26. Quick Start Signal Tap File Templates

Template	Summary Description
Default	The most basic and compact setup that is suitable for many debugging needs
Default with Hidden Hierarchy and Data Log	The same setup as the Default template, with additional Hierarchy Display and Data Log windows for trigger condition setup.
State-Based Trigger Flow Control	Starts with three conditions setup to replicate the basic sequential trigger flow control.
Conditional Storage Qualifier	Enables the **Conditional** storage qualifier and **Basic OR** condition. This setup provides a versatile storage qualifier condition expression.
Transitional Storage Qualifier	Enables the **Transitional** storage qualifier. The **Transitional** storage qualifier simply detects changes in data.
Start-Stop Storage Qualifier	Enables the **Start/Stop** storage qualifier and the **Basic OR** condition. Provides two conditions to frame the data.
State-Based Storage Qualifier	Provides more sophisticated qualification conditions for use with state machine expressions. You must use the State-Based Storage Qualifier template in conjunction with the state-based trigger flow control

continued...

Template	Summary Description
Input Port Storage Qualifier	Enables the **Input port** storage qualifier to provide total control of the storage qualifier condition by supporting development of custom logic outside of the Signal Tap logic hierarchy.
Trivial Advanced Trigger Condition	Enables the **Advanced** trigger condition. The **Advanced** condition provides the most flexibility to express complex conditions. The **Advanced** trigger condition scales from a simple wire to the most complex logical expression. This template starts with the simplest condition.
Trigger Position Defined Using Sample Count	Supports specifying an exact number of samples to store after the trigger position, using the **State-Based Trigger Flow Control** template as a reference.
Cross-triggering Between STP Instances	Enables "Cross-triggering by using the **Trigger out** from one instance as the **Trigger in** of another instance, when using multiple Signal Tap instances.
Setup for Incremental Compilation	Specifies a fixed input width for signal inputs. This technique allows efficient incremental compilation by reducing the amount of Signal Tap logic change, and by adding only post-fit nodes to tap.
Define Trigger Condition in RTL	Supports defining a custom trigger condition in the RTL language of your choice.

Table 27. Standard Interface Signal Tap File Templates

Template	Summary Description
Capture Avalon Memory Mapped Transactions	Allows you to use the storage qualifier feature to store only meaningful Avalon memory-mapped interface transactions.
Simple Avalon Streaming Interface Bus Performance Analysis	Supports recording of event time for analysis of the data packet flow in an Avalon streaming interface.
Use Counters in the State-based Flow Control to Collect Stats	Use counters to track of the number of packets produced (`pkt_counter`), number of data beats produced (`pkt_beat_counter`), and number of data beats consumed (`stream_beat_counter`).

Table 28. State-Based Triggering Design Flow Examples Signal Tap File Templates

Template	Setup Description
Trigger on an Event Absent for Greater Than or Equal to 5 Clock Cycles	Requires setup of one basic trigger condition in the **Setup** tab to the value that you want.
Trigger on Event Absent for Less Than 5 Clock Cycles	Requires setup of one basic trigger condition in the **Setup** tab to the value that you want.
Trigger on 5th Occurrence of a Group Value	Requires setup of one basic trigger condition in the **Setup** tab to the value that you want.
Trigger on the 5th Transition of a Group Value	Requires setup of an edge-sensitive trigger condition to detect all bus transitions to the desired group value. Requires edge detection for any data bus bit logically ANDed with a comparison to the desired group value. An advanced trigger condition is necessary in this case.
Trigger After Condition1 is Followed by Condition2	Requires setup of three basic trigger conditions in the **Setup** tab to the values you specify. The first two trigger conditions are set to the desired group values. The third trigger condition is set to capture some type of idle transaction across the bus between the first and second conditions.
Trigger on Condition1 Immediately Followed by Condition2	Requires setup of two basic trigger conditions in the **Setup** tab to the group values that you want.
Trigger on Condition2 Not Occurring Between Condition1 and Condition3	Requires setup of three basic trigger conditions in the **Setup** tab to the group values that you want.

continued...

Template	Setup Description
Trigger on the 5th Consecutive Occurrence of Condition1	Requires setup of one basic trigger condition in the **Setup** tab to the value you want.
Trigger After a Violation of Sequence From Condition1 To Condition4	Requires setup of four basic trigger conditions to the sequence values that you want.
Trigger on a Sequence of Edges	Requires setup of three edge-sensitive basic trigger conditions for the sequence that you want.
Trigger on Condition1 Followed by Condition2 After 5 Clock Cycles	Requires setup of two basic trigger conditions to the group values that you want.
Trigger on Condition1 Followed by Condition2 Within 5 Samples	Requires setup of two basic trigger conditions to the group values that you want.
Trigger on Condition1 Not Followed by Condition2 Within 5 Samples	Requires setup of two basic trigger conditions to the group values that you want.
Trigger After 5 Consecutive Transitions	Requires setup of a trigger condition to capture any transition activity on the monitored bus. This example requires an Advanced trigger condition because the example requires an OR condition.
Trigger When Condition1 Occurs Less Than 5 Times Between Condition2 and Condition3	Requires setup of three edge-sensitive trigger conditions, with each trigger condition containing a comparison to the desired group value.

2.13. Running the Stand-Alone Version of Signal Tap

You can optionally install a stand-alone version of the Signal Tap logic analyzer, rather than using the Signal Tap logic analyzer integrated with the Intel Quartus Prime software.

The stand-alone version of Signal Tap is particularly useful in a lab environment that lacks a suitable workstation for a complete Intel Quartus Prime installation, or lacks a full Intel Quartus Prime software license.

The standalone version of the Signal Tap logic analyzer includes and requires use of the Intel Quartus Prime stand-alone Programmer, which is also available from the Download Center for FPGAs.

2.14. Signal Tap Scripting Support

The Intel Quartus Prime software supports automation of Signal Tap controls in a Tcl scripting environment, or with the `quartus_stp` executable. For detailed information about scripting command options, refer to the Intel Quartus Prime command-line and Tcl API help by typing `quartus_sh --qhelp` at the command prompt.

Related Information

- Tcl Scripting
  In *Intel Quartus Prime Pro Edition User Guide: Scripting*
- Command Line Scripting
  In *Intel Quartus Prime Pro Edition User Guide: Scripting*

2.14.1. Signal Tap Command-Line Options

You can use the following options with the `quartus_stp` executable:

Table 29. quartus_stp Command-Line Options

Option	Usage	Description
`--stp_file` *<stp_filename>*	Required	Specifies the name of the `.stp` file.
`--enable`	Optional	Sets the ENABLE_SIGNALTAP option to `ON` in the project's `.qsf` file, so the Signal Tap logic analyzer runs in the next compilation. If you omit this option, the Intel Quartus Prime software uses the current value of ENABLE_SIGNALTAP in the `.qsf` file. Writes subsequent Signal Tap assignments to the `.stp` that appears in the `.qsf` file. If the `.qsf` file does not specify a `.stp` file, you must use the `--stp_file` option.
`--disable`	Optional	Sets the ENABLE_SIGNALTAP option to `OFF` in the project's `.qsf` file, so the Signal Tap logic analyzer does not in the next compilation. If you omit the `--disable` option, the Intel Quartus Prime software uses the current value of ENABLE_SIGNALTAP in the `.qsf` file.

2.14.2. Data Capture from the Command Line

The `quartus_stp` executable supports a Tcl interface that allows you to capture data without running the Intel Quartus Prime GUI.

Note: You cannot execute Signal Tap Tcl commands from within the Tcl console in the Intel Quartus Prime software.

To execute a Tcl script containing Signal Tap logic analyzer Tcl commands, use:

```
quartus_stp -t <Tcl file>
```

Example 2. Continuously Capturing Data

This excerpt shows commands you can use to continuously capture data. Once the capture meets trigger condition, the Signal Tap logic analyzer starts the capture and stores the data in the data log.

```
#  Open Signal Tap session
open_session -name stp1.stp

###  Start acquisition of instances auto_signaltap_0 and
###  auto_signaltap_1 at the same time

# Calling run_multiple_end starts all instances
run_multiple_start

run -instance auto_signaltap_0 -signal_set signal_set_1 -trigger \
     trigger_1 -data_log log_1 -timeout 5
run -instance auto_signaltap_1 -signal_set signal_set_1 -trigger \
     trigger_1 -data_log log_1 -timeout 5

run_multiple_end

# Close Signal Tap session
close_session
```

Related Information

::quartus::stp
In *Intel Quartus Prime Help*

2.15. Signal Tap File Version Compatibility

If you open a .stp file created in a previous version of the Intel Quartus Prime software in a newer version of the software, you can no longer open that .stp file in the previous version of the Intel Quartus Prime software.

If you open an Intel Quartus Prime project that includes a .stp file from a previous version of the software in a later version of the Intel Quartus Prime software, the software may require you to update the .stp configuration file before you can compile the project. Update the configuration file by simply opening the .stp in the Signal Tap logic analyzer GUI. If configuration update is required, Signal Tap confirms that you want to update the .stp to match the current version of the Intel Quartus Prime software.

Note: The Intel Quartus Prime Pro Edition software uses a new methodology for settings and assignments. For example, Signal Tap assignments include only the instance name, not the entity:instance name. Refer to *Migrating to Intel Quartus Prime Pro Edition* for more information about migrating existing Signal Tap files (.stp) to Intel Quartus Prime Pro Edition.

Related Information

Migrating to Intel Quartus Prime Pro Edition, Intel Quartus Prime Pro Edition User Guide: Getting Started

2.16. Design Debugging with the Signal Tap Logic Analyzer Revision History

The following revision history applies to this chapter:

Document Version	Intel Quartus Prime Version	Changes
2021.10.13	21.3	• Added *Recompiling Only Signal Tap Changes* topic. • Changed title of *Prevent Changes Requiring Recompilation* to *Preventing Changes that Require Full Recompilation* and revised figures.
2021.10.04	21.3	• Updated *Signal Tap Logic Analyzer Introduction* with *Signal Tap Logic Analyzer and Simulator Integration* section. • Added description of Autorun mode to *Managing Signal Tap Instances* topic. • Added new *Adding Simulator-Aware Signal Tap Nodes* topic. • Added new *Add Simulator Aware Node Finder Settings* topic. • Added new *Signal Tap and Simulator Integration* topic. • Added new *Generating a Simulation Testbench from Signal Tap Data* topic. • Added new *Create Simulation Testbench Dialog Box Settings* topic. • Revised *Preserving Signals for Monitoring and Debugging* topic for latest techniques and links to other resources. • Revised *Adding Pre-Synthesis or Post-Fit Nodes* for latest techniques and links to other resources.

continued...

Document Version	Intel Quartus Prime Version	Changes
		• Added new *Changing the Post-Fit Signal Tap Target Nodes* topic. • Updated *Adding Pre-Synthesis or Post-Fit Nodes* topic for preserve for debug filters. • Added details about SOF Manager to *Ensure Compatibility Between .stp and .sof Files* topic.
2020.09.28	20.3	• Revised "Signal Tap Logic Analyzer Introduction" for screenshot and details about role of Signal Tap Intel FPGA IP. • Revised graphic and wording in "Signal Tap Hardware and Software Requirements" topic. • Revised wording and link to download in "Running the Stand-Alone Version of Signal Tap." • Updated flow diagram and added links to retitled "Signal Tap Debugging Flow" topic. • Retitled "Add the Signal Tap Logic Analyzer to Your Design" to "Step 1: Add the Signal Tap Logic Analyzer to the Project," and referenced new template and added links to next steps. • Added "Creating a Signal Tap Instance with the Signal Tap GUI" topic. • Added new "Signal Tap File Templates" topic. • Added new "Creating a Signal Tap Instance by HDL Instantiation" topic. • Added new "Signal Tap Intel FPGA IP Parameters" topic. • Retitled "Configure the Signal Tap Logic Analyzer" to "Step 2: Configure the Signal Tap Logic Analyzer," and referenced new template and added links to next steps. • Enhanced description in Step 5: Run the Signal Tap Logic Analyzer" topic. • Revised "Adding Signals to the Signal Tap Logic Analyzer" to add detailed steps and screenshot. • Retitled and revised "Adding Nios II Processor Signals" to reflect there is only one plug-in in Intel Quartus Prime Pro Edition. • Revised "Disabling or Enabling Signal Tap Instances" and added screenshot. • Replaced outdated links to AN446 with links to AN845. • Revised headings and steps in "Debugging Block-Based Designs with Signal Tap" section. • Retitled "Debugging Imported Snapshots" to "Compiler Snapshots and Signal Tap Debugging". • Retitled "Backward Compatibility" to "Signal Tap File Version Compatibility." • Removed incorrect statement about debugging multiple designs from "Step 4: Program the Target Hardware" topic. • Removed reference to obsolete resource checking function from "Ensure Compatibility Between STP and SOF Files" topic. • Removed obsolete "Remote Debugging Using the Signal Tap Logic Analyzer" section. • Removed obsolete "Estimating FPGA Resources" topic.
2019.06.11	18.1.0	Added more explanation to Figure 64 on page 76 about continuous and input mode.
2019.05.01	18.1.0	In *Adding Signals with a Plug-In* topic, removed outdated information from step 1 about turning on **Create debugging nodes for IP cores**.
2018.09.24	18.1.0	• Added content about debugging designs in block-based flows. • Renamed topic: *Untappable Signals* to *Signals Unavailable for Signal Tap Debugging*.
2018.08.07	18.0.0	Reverted document title to *Debug Tools User Guide: Intel Quartus Prime Pro Edition*.
2018.07.30	18.0.0	Updated Partial Reconfiguration sections to reflect changes in the PR flow.

continued...

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0.0	• Added note stating Signal Tap IP not optimized for Stratix 10 Devices. • Moved information about debug fabric on PR designs to the *System Debugging Tools Overview* chapter. • Removed restrictions of Rapid Recompile support for Intel Stratix 10 devices.
2017.11.06	17.1.0	• Added support for Incremental Routing in Intel Stratix 10 devices. • Removed unsupported FSM auto detection. • Clarified information about the Data Log Pane. • Updated Figure: Data Log and renamed to Simple Data Log. • Added Figure: Accessing the Advanced Trigger Condition Tab. • Removed outdated information about command-line flow.
2017.05.08	17.0.0	• Added: Open Standalone Signal Tap Logic Analyzer GUI. • Added: Debugging Partial Reconfiguration Designs Using Signal Tap Logic Analyzer. • Updated figures on Create Signal Tap File from Design Instance(s).
2016.10.31	16.1.0	• Implemented Intel rebranding. • Added: Create SignalTap II File from Design Instance(s). • Removed reference to unsupported Talkback feature.
2016.05.03	16.0.0	• Added: Specifying the Pipeline Factor • Added: Comparison Trigger Conditions
2015.11.02	15.1.0	• Changed instances of *Quartus II* to *Intel Quartus Prime*. • Updated content to reflect SignalTap II support in Intel 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.
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.

continued...

Document Version	Intel Quartus Prime Version	Changes
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 II 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 II software version 8.1 release: • Added new section "Using the Storage Qualifier Feature" on page 14–25 • Added description of `start_store` and `stop_store` commands in section "Trigger Condition Flow Control" on page 14–36 • Added new section "Runtime Reconfigurable Options" on page 14–63
May 2008	8.0.0	Updated for the Quartus II 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

3. Quick Design Verification with Signal Probe

This chapter describes a technique that provides debug access to internal device signals without affecting the design.

The Signal Probe feature in the Intel Quartus Prime Pro Edition software allows you to route an internal node to a top-level I/O. When you start with a fully routed design, you can select and route debugging signals to I/O pins that you previously reserve or are currently unused.

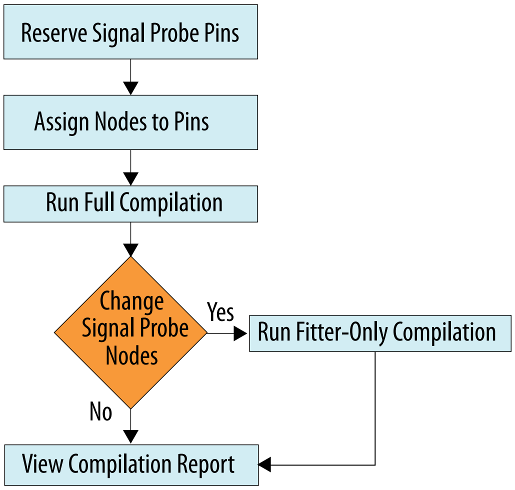
Related Information

System Debugging Tools Overview on page 6

3.1. Signal Probe Debugging Flow

Use the following flow to add Signal Probe debugging and verification capabilities to your design:

Figure 95. Signal Probe Debugging Flow


Reserve Signal Probe Pins
Assign Nodes to Pins
Run Full Compilation
Change Signal Probe Nodes
Yes
Run Fitter-Only Compilation
No
View Compilation Report

Step 1: Reserve Signal Probe Pins on page 123

Step 2: Assign Nodes to Signal Probe Pins on page 123

Step 3: Connect the Signal Probe Pin to an Output Pin on page 123

Step 4: Compile the Design on page 124

(Optional) Step 5: Modify the Signal Probe Pins Assignments on page 124

Step 6: Run Fitter-Only Compilation on page 124

Step 7: Check Connection Table in Fitter Report on page 125

3.1.1. Step 1: Reserve Signal Probe Pins

You must first create and reserve a pin for Signal Probe with a Tcl command:

```
set_global_assignment –name CREATE_SIGNALPROBE_PIN <pin_name>
```

pin_name Specifies the name of the Signal Probe pin.

Optionally, you can assign locations for the Signal Probe pins. If you do not assign a location, the Fitter places the pins automatically.

Note: If from the onset of the debugging process you know which internal signals you want to route, you can reserve pins and assign nodes before compilation. This early assignment removes the recompilation step from the flow.

Example 3. Tcl Command to Reserve Signal Probe Pins

```
set_global_assignment -name CREATE_SIGNALPROBE_PIN wizard
set_global_assignment -name CREATE_SIGNALPROBE_PIN probey
```

Related Information

Constraining Designs with Tcl Scripts
In *Intel Quartus Prime Pro Edition User Guide: Design Constraints*

3.1.2. Step 2: Assign Nodes to Signal Probe Pins

You can assign any node in the post-compilation netlist to a Signal Probe pin. In the Intel Quartus Prime software, click **View ➤ Node Finder**, and filter by **Signal Tap: post-fitting** to view the nodes you can route.

You specify the node that connects to a Signal Probe pin with a Tcl command:

```
set_instance_assignment –name CONNECT_SIGNALPROBE_PIN <pin_name> \
     –to <node_name>
```

pin_name Specifies the name of the Signal Probe pin that connects to the node.

node_name Specifies the full hierarchy path of the node you want to route.

Example 4. Tcl Commands to Connect Pins to Internal Nodes

```
# Make assignments to connect nodes of interest to pins
set_instance_assignment -name CONNECT_SIGNALPROBE_PIN wizard -to sprobe_me1
set_instance_assignment -name CONNECT_SIGNALPROBE_PIN probey -to sprobe_me2
```

3.1.3. Step 3: Connect the Signal Probe Pin to an Output Pin

Once you reserve pins and assign internal nodes to the Signal Probe pins, you must connect the Signal Probe pin to an external output pin.

Example 5. Tcl Command to Specify Signal Probe Pin Assignment

```
set_instance_assignment –name CONNECT_SIGNALPROBE_PIN <pin_name> –to <node_name>
```

3.1.4. Step 4: Compile the Design

Perform a full compilation of the design. You can use Intel Quartus Prime software GUI, a command line executable, or the following Tcl command to start the Compiler

Example 6. Tcl Command to Compile the Design

```
execute_flow -compile
```

At this point in the design flow, you can determine the nodes that you want to debug.

Related Information

Design Compilation
In *Intel Quartus Prime Pro Edition User Guide: Compiler*

3.1.5. (Optional) Step 5: Modify the Signal Probe Pins Assignments

As long as you reserve the pins (with CREATE_SIGNALPROBE_PIN) before running full compilation, you can optionally add or modify the node that connects to a reserved Signal Probe pin (with CONNECT_SIGNALPROBE_PIN) without rerunning a full compilation. Rather, you can run a Fitter-only compilation to implement the Signal Probe pin assignment change.

Note: If you modify the physical I/O pin assignments with (with CREATE_SIGNALPROBE_PIN) after running compilation, you must rerun full compilation to implement those changes before using Signal Probe.

Example 7. Tcl Command to Specify Signal Probe Pin Assignment

```
set_instance_assignment –name CONNECT_SIGNALPROBE_PIN <pin_name> –to <node_name>
```

3.1.6. Step 6: Run Fitter-Only Compilation

After re-assigning nodes to the Signal Probe pins, you can run a Fitter-only compilation (using --recompile) to implement the post-fit change without rerunning a full compilation.

Example 8. Tcl Command to Run Fitter-Only Compile

```
# Run the fitter with --recompile to preserve timing
# and quickly connect the Signal Probe pins
execute_module -tool fit -args {--recompile}
```

After recompilation, you are ready to program the device and debug the design.

Related Information

- Using Rapid Recompile
  In *Intel Quartus Prime Pro Edition User Guide: Compiler*
- Using the ECO Compilation Flow, Intel Quartus Prime Pro Edition User Guide: Design Optimization

3.1.7. Step 7: Check Connection Table in Fitter Report

When you compile a design with Signal Probe pins, Compiler generates a connection report showing the connection status to Signal Probe pins.

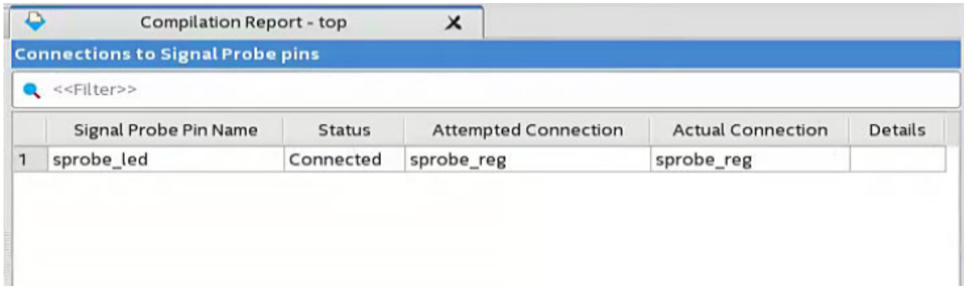
To view this report, click **Processing ➤ Compilation Report**, open the **Fitter ➤ In-System Debugging** folder, and click **Connections to Signal Probe pins**.

The **Status** column indicates whether or not the routing attempt from the nodes to the Signal Probe pins is successful.

Table 30. Status of Signal Probe Connection

Status	Description
Connected	Routing succeeded.
Unconnected	Routing did not succeed. Possible reasons are: • Node belongs to an I/O cell or another hard IP, thus cannot be routed. • Node hierarchy path does not exist in the design. • Node is not **Signal Tap: post-fitting**.

Example 9. Connections to Signal Probe Pins in the Compilation Report



	Signal Probe Pin Name	Status	Attempted Connection	Actual Connection	Details
1	sprobe_led	Connected	sprobe_reg	sprobe_reg	

Alternatively, you can find the Signal Probe connection information in the Fitter report file (*<project_name>*`.fit.rpt`).

Example 10. Connections to Signal Probe Pins in top.fit.rpt

```
+--------------------------------------------------------------------------------
-+
; Connections to Signal Probe
pins                                                  ;
+--------------------------------------------------------------------------------
-+
Signal Probe Pin Name : probey
Status                : Connected
Attempted Connection  : sprobe_me2
Actual Connection     : sprobe_me2
Details               :

Signal Probe Pin Name : wizard
Status                : Connected
Attempted Connection  : sprobe_me1
Actual Connection     : sprobe_me1
Details               :
+--------------------------------------------------------------------------------
-+
```

Related Information

- Signals Unavailable for Signal Tap Debugging on page 50
- Text-Based Report Files
  In *Intel Quartus Prime Pro Edition User Guide: Scripting*

3.2. Quick Design Verification with Signal Probe Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.10.04	21.3	• Removed references to obsolete Rapid Recompile feature. • Updated *Signal Probe Debugging Flow* topic for new optional step and added flow diagram. • Added step numbers to tasks in flow to emphasize order of operations. • Added *(Optional) Step 4: Modify the Signal Probe Pins Assignments* topic. • Revised wording in *Step 5: Run Fitter-Only Compilation*. • Revised screenshot and wording in *Step 6: Check Connection Table in Fitter Report*. • Added new *Step 3: Connect the Signal Probe Pin to an Output Pin* topic.
2018.05.07	18.0.0	Initial release for Intel Quartus Prime Pro Edition software.

4. In-System Debugging Using External Logic Analyzers

4.1. About the Intel Quartus Prime Logic Analyzer Interface

The Intel Quartus Prime Logic Analyzer Interface (LAI) allows you to use an external logic analyzer and a minimal number of Intel-supported device I/O pins to examine the behavior of internal signals while your design is running at full speed on your Intel-supported device.

The LAI connects a large set of internal device signals to a small number of output pins. You can connect these output pins to an external logic analyzer for debugging purposes. In the Intel Quartus Prime LAI, the internal signals are grouped together, distributed to a user-configurable multiplexer, and then output to available I/O pins on your Intel-supported device. Instead of having a one-to-one relationship between internal signals and output pins, the Intel Quartus Prime LAI enables you to map many internal signals to a smaller number of output pins. The exact number of internal signals that you can map to an output pin varies based on the multiplexer settings in the Intel Quartus Prime LAI.

Note: The term “logic analyzer” when used in this document includes both logic analyzers and oscilloscopes equipped with digital channels, commonly referred to as mixed signal analyzers or MSOs.

The LAI does not support Hard Processor System (HPS) I/Os.

Related Information

Device Support Center

4.2. Choosing a Logic Analyzer

The Intel Quartus Prime software offers the following two general purpose on-chip debugging tools for debugging a large set of RTL signals from your design:

- The Signal Tap Logic Analyzer
- An external logic analyzer, which connects to internal signals in your Intel-supported device by using the Intel Quartus Prime LAI

Table 31. Comparing the Signal Tap Logic Analyzer with the Logic Analyzer Interface

Feature	Description	Recommended Logic Analyzer
Sample Depth	You have access to a wider sample depth with an external logic analyzer. In the Signal Tap Logic Analyzer, the maximum sample depth is set to	LAI

continued...

Feature	Description	Recommended Logic Analyzer
	128 Kb, which is a device constraint. However, with an external logic analyzer, there are no device constraints, providing you a wider sample depth.	
Debugging Timing Issues	Using an external logic analyzer provides you with access to a "timing" mode, which enables you to debug combined streams of data.	LAI
Performance	You frequently have limited routing resources available to place and route when you use the Signal Tap Logic Analyzer with your design. An external logic analyzer adds minimal logic, which removes resource limits on place-and-route.	LAI
Triggering Capability	The Signal Tap Logic Analyzer offers triggering capabilities that are comparable to external logic analyzers.	LAI or Signal Tap
Use of Output Pins	Using the Signal Tap Logic Analyzer, no additional output pins are required. Using an external logic analyzer requires the use of additional output pins.	Signal Tap
Acquisition Speed	With the Signal Tap Logic Analyzer, you can acquire data at speeds of over 200 MHz. You can achieve the same acquisition speeds with an external logic analyzer; however, you must consider signal integrity issues.	Signal Tap

Related Information

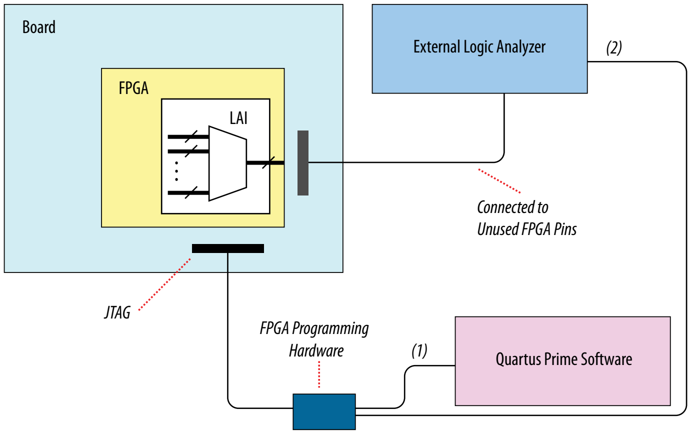
System Debugging Tools Overview on page 6

4.2.1. Required Components

To perform analysis using the LAI you need the following components:

- Intel Quartus Prime software version 15.1 or later
- The device under test
- An external logic analyzer
- An Intel FPGA communications cable
- A cable to connect the Intel-supported device to the external logic analyzer

Figure 96. LAI and Hardware Setup

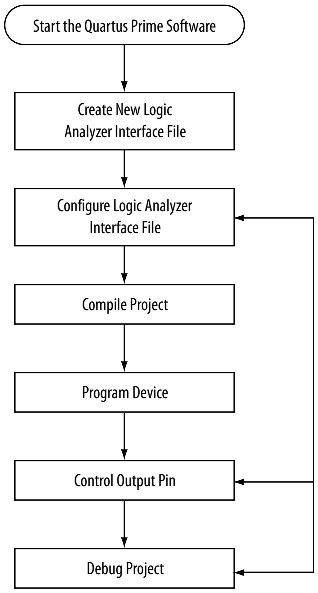


Notes to figure:

1. Configuration and control of the LAI using a computer loaded with the Intel Quartus Prime software via the JTAG port.
2. Configuration and control of the LAI using a third-party vendor logic analyzer via the JTAG port. Support varies by vendor.

4.3. Flow for Using the LAI

Figure 97. LAI Workflow



Notes to figure:

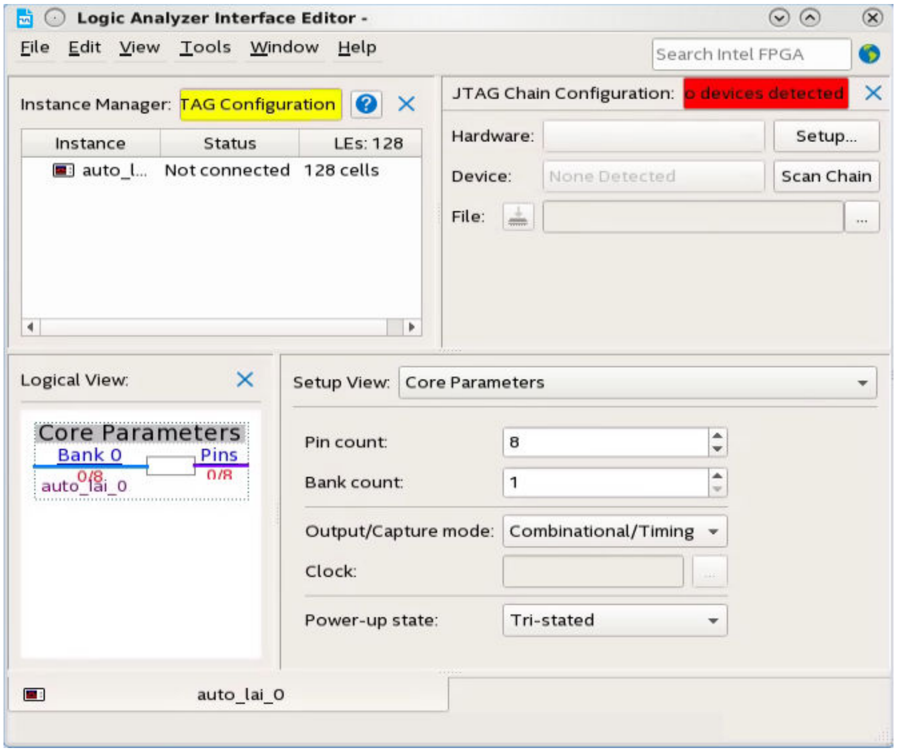
1. Configuration and control of the LAI using a computer loaded with the Intel Quartus Prime software via the JTAG port.
2. Configuration and control of the LAI using a third-party vendor logic analyzer via the JTAG port. Support varies by vendor.

4.3.1. Defining Parameters for the Logic Analyzer Interface

The **Logic Analyzer Interface Editor** allows you to define the parameters of the LAI.

- Click **Tools ➤ Logic Analyzer Interface Editor**.

Figure 98. Logic Analyzer Interface Editor



Instance Manager: TAG Configuration

Instance	Status	LEs: 128
auto_l...	Not connected	128 cells

JTAG Chain Configuration: o devices detected

Hardware:

Device: None Detected

File:

Setup View: Core Parameters

Pin count: 8

Bank count: 1

Output/Capture mode: Combinational/Timing

Clock:

Power-up state: Tri-stated

auto_lai_0

- In the **Setup View** list, select **Core Parameters**.
- Specify the parameters of the LAI instance.

Related Information

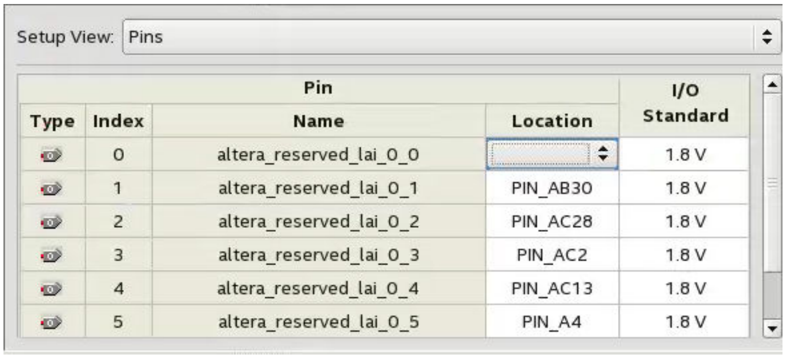
LAI Core Parameters on page 133

4.3.2. Mapping the LAI File Pins to Available I/O Pins

To assign pin locations for the LAI:

1. Select **Pins** in the **Setup View** list

Figure 99. Mapping LAI file Pins



Setup View: Pins

Pin				I/O Standard
Type	Index	Name	Location	
⚙	0	altera_reserved_lai_0_0		1.8 V
⚙	1	altera_reserved_lai_0_1	PIN_AB30	1.8 V
⚙	2	altera_reserved_lai_0_2	PIN_AC28	1.8 V
⚙	3	altera_reserved_lai_0_3	PIN_AC2	1.8 V
⚙	4	altera_reserved_lai_0_4	PIN_AC13	1.8 V
⚙	5	altera_reserved_lai_0_5	PIN_A4	1.8 V

2. Double-click the **Location** column next to the reserved pins in the **Name** column, and select a pin from the list.
3. Right-click the selected pin and locate in the Pin Planner.

Related Information

Managing Device I/O Pins
In *Intel Quartus Prime Pro Edition User Guide: Design Constraints*

4.3.3. Mapping Internal Signals to the LAI Banks

After specifying the number of banks to use in the **Core Parameters** settings page, you must assign internal signals for each bank in the LAI.

1. Click the **Setup View** arrow and select **Bank n** or **All Banks**.
2. To view all the bank connections, click **Setup View** and then select **All Banks**.
3. Before making bank assignments, right click the Node list and select **Add Nodes** to open the **Node Finder**.
4. Find the signals that you want to acquire.
5. Drag the signals from the **Node Finder** dialog box into the bank **Setup View**.

   When adding signals, use **Signal Tap: pre-synthesis** for non-incrementally routed instances and **Signal Tap: post-fitting** for incrementally routed instances

   As you continue to make assignments in the bank **Setup View**, the schematic of the LAI in the **Logical View** pane begins to reflect the changes.
6. Continue making assignments for each bank in the **Setup View** until you add all the internal signals that you want to acquire.

Related Information

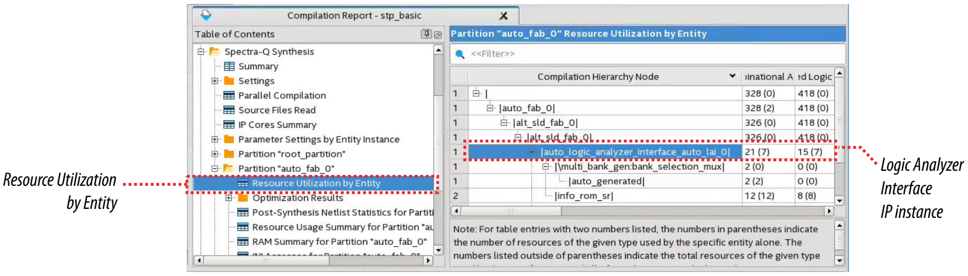
Node Finder Command
In *Intel Quartus Prime Help*

4.3.4. Compiling Your Intel Quartus Prime Project

After you save your `.lai` file, a dialog box prompts you to enable the Logic Analyzer Interface instance for the active project. Alternatively, you can define the `.lai` file your project uses in the **Global Project Settings** dialog box. After specifying the name of your `.lai` file, compile your project.

To verify the Logic Analyzer Interface is properly compiled with your project, open the **Compilation Report** tab and select Resource Utilization by Entity, nested under Partition "auto_fab_0". The LAI IP instance appears in the Compilation Hierarchy Node column, nested under the internal module of `auto_fab_0`

Figure 100. LAI Instance in Compilation Report


Compilation Report - stp_basic
Table of Contents
Spectra-Q Synthesis
Summary
Settings
Parallel Compilation
Source Files Read
IP Cores Summary
Parameter Settings by Entity Instance
Partition "root_partition"
Partition "auto_fab_0"
Resource Utilization by Entity
Optimization Results
Post-Synthesis Netlist Statistics for Partiti
Resource Usage Summary for Partition "au
RAM Summary for Partition "auto_fab_0"
<<Filter>>

Partition "auto_fab_0" Resource Utilization by Entity

	Compilation Hierarchy Node	inational A	ed Logic
1	|	328 (0)	418 (0)
1	|auto_fab_0|	328 (2)	418 (0)
1	|alt_sld_fab_0|	326 (0)	418 (0)
1	|alt_sld_fab_0|	326 (0)	418 (0)
1	|auto_logic_analyzer_interface_auto_lai_0|	21 (7)	15 (7)
1	|\multi_bank_gen:bank_selection_mux|	2 (0)	0 (0)
1	|auto_generated|	2 (2)	0 (0)
2	|info_rom_sr|	12 (12)	8 (8)

Note: For table entries with two numbers listed, the numbers in parentheses indicate the number of resources of the given type used by the specific entity alone. The numbers listed outside of parentheses indicate the total resources of the given type

Resource Utilization by Entity

Logic Analyzer Interface IP instance

4.3.5. Programming Your Intel-Supported Device Using the LAI

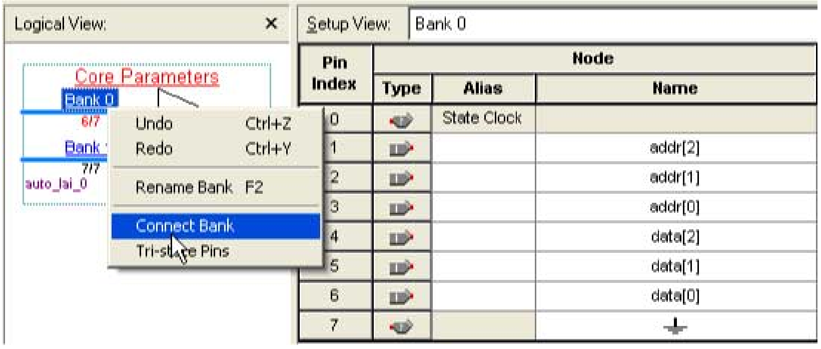
After compilation completes, you must configure your Intel-supported device before using the LAI.

You can use the LAI with multiple devices in your JTAG chain. Your JTAG chain can also consist of devices that do not support the LAI or non-Intel, JTAG-compliant devices. To use the LAI in more than one Intel-supported device, create an `.lai` file and configure an `.lai` file for each Intel-supported device that you want to analyze.

4.4. Controlling the Active Bank During Runtime

When you have programmed your Intel-supported device, you can control which bank you map to the reserved `.lai` file output pins. To control which bank you map, in the schematic in the Logical View, right-click the bank and click **Connect Bank**.

Figure 101. 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	◀		⏚

4.4.1. Acquiring Data on Your Logic Analyzer

To acquire data on your logic analyzer, you must establish a connection between your device and the external logic analyzer. For more information about this process and for guidelines about how to establish connections between debugging headers and logic analyzers, refer to the documentation for your logic analyzer.

4.5. LAI Core Parameters

The table lists the LAI file core parameters:

Table 32. LAI File Core Parameters

Parameter	Range Value	Description
Pin Count	1 - 255	Number of pins dedicated to the LAI. You must connect the pins to a debug header on the board. Within the device, The Compiler maps each pin to a user-configurable number of internal signals.
Bank Count	1 - 255	Number of internal signals that you want to map to each pin. For example, a **Bank Count** of 8 implies that you connect eight internal signals to each pin.
Output/Capture Mode		Specifies the acquisition mode. The two options are: • **Combinational/Timing**—This acquisition mode uses the external logic analyzer's internal clock to determine when to sample data. This acquisition mode requires you to manually determine the sample frequency to debug and verify the system, because the data sampling is asynchronous to the Intel-supported device. 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 external logic analyzer's data sheet. • **Registered/State**—This acquisition mode determines when to sample from a signal on the system under test. Consequently, the data samples are synchronous with the Intel-supported device. The **Registered/State** mode provides a functional view of the Intel-supported device while it is running. This mode is effective when you verify the functionality of the design.
Clock		Specifies the sample clock. You can use any signal in the design as a sample clock. However, for best results, use a clock with an operating frequency fast enough to sample the data that you want to acquire. *Note:* The **Clock** parameter is available only when **Output/Capture Mode** is set to **Registered State**.
Power-Up State		Specifies the power-up state of the pins designated for use with the LAI. You can select tri-stated for all pins, or selecting a particular bank that you enable.

Related Information

Defining Parameters for the Logic Analyzer Interface on page 130

4.6. In-System Debugging Using External Logic Analyzers Revision History

The following revision history applies to this chapter:

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0.0	• Moved list of LAI File Core Parameters from *Configuring the File Core Parameters* to its own topic, and added links.
2017.05.08	17.0.0	• Updated *Compiling Your Intel Quartus Prime Project* • Updated figure: LAI Instance in Compilation Report.

continued...

Document Version	Intel Quartus Prime Version	Changes
2016.10.31	16.1.0	- Implemented Intel rebranding.
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
June 2014	14.0.0	- Dita conversion - Added limitation about HPS I/O support
June 2012	12.0.0	Removed survey link
November 2011	10.1.1	Changed to new document template
December 2010	10.1.0	- Minor editorial updates - Changed to new document template
August 2010	10.0.1	Corrected links
July 2010	10.0.0	- Created links to the Intel Quartus Prime Help - Editorial updates - Removed Referenced Documents section
November 2009	9.1.0	- Removed references to APEX devices - Editorial updates
March 2009	9.0.0	- Minor editorial updates - Removed Figures 15–4, 15–5, and 15–11 from 8.1 version
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content
May 2008	8.0.0	- Updated device support list on page 15–3 - Added links to referenced documents throughout the chapter - Added “Referenced Documents” - Added reference to *Section V. In-System Debugging* - Minor editorial updates

5. In-System Modification of Memory and Constants

The Intel Quartus Prime In-System Memory Content Editor (ISMCE) allows to view and update memories and constants at runtime through the JTAG interface. By testing changes to memory contents in the FPGA while the design is running, you can identify, test, and resolve issues.

The ability to read data from memories and constants can help you identify the source of problems, and the write capability allows you to bypass functional issues by writing expected data.

When you use the In-System Memory Content Editor in conjunction with the Signal Tap logic analyzer, you can view and debug your design in the hardware lab.

Related Information

- System Debugging Tools Overview on page 6
- Design Debugging with the Signal Tap Logic Analyzer on page 28

5.1. IP Cores Supporting ISMCE

In Intel Arria 10 and Intel Stratix 10 device families, you can use the ISMCE in RAM: 1 PORT and the ROM: 1 PORT IP Cores.

Note: To use the ISMCE tool with designs migrated from older devices to Intel Stratix 10 devices, replace instances of the altsyncram Intel FPGA IP with the altera_syncram Intel FPGA IP.

Related Information

- Intel Stratix 10 Embedded Memory IP Core References
  In *Intel Stratix 10 Embedded Memory User Guide*
- About Embedded Memory IP Cores
  In *Embedded Memory (RAM: 1-PORT, RAM: 2-PORT, ROM: 1-PORT, and ROM: 2-PORT) User Guide*
- Intel FPGA IP Cores/LPM
  In *Intel Quartus Prime Help*

5.2. Debug Flow with the In-System Memory Content Editor

To debug a design with the In-System Memory Content Editor:

1. Identify the memories and constants that you want to access at runtime.
2. Specify in the design the memory or constant that must be run-time modifiable.
3. Perform a full compilation.

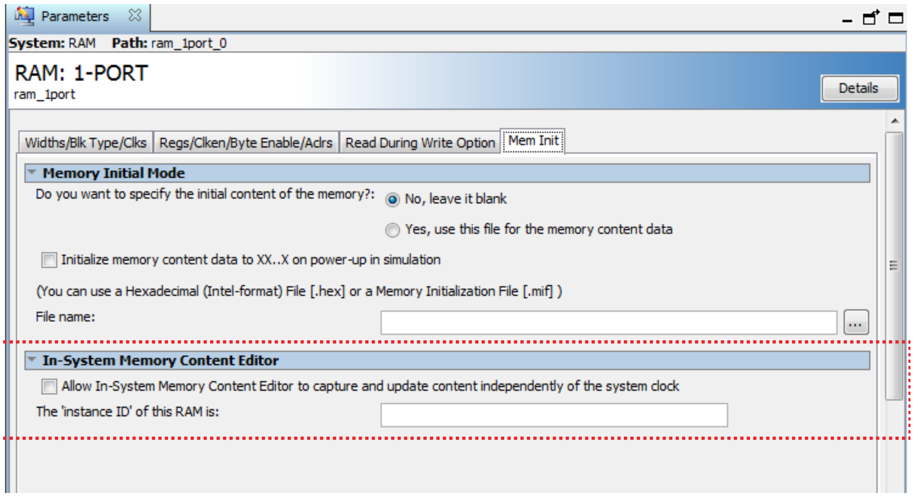
4. Program the device.
5. Launch the In-System Memory Content Editor.
   The In-System Memory Content Editor retrieves all instances of run-time configurable memories and constants by scanning the JTAG chain and sending a query to the device selected in the JTAG Chain Configuration pane.
6. Modify the values of the memories or constants, and check the results.

   For example, if a parity bit in a memory is incorrect, you can use the In-System Memory Content Editor to write the correct parity bit values into the RAM, allowing the system to continue functioning. To check the error handling functionality of a design, you can intentionally write incorrect parity bit values into the RAM.

5.3. Enabling Runtime Modification of Instances in the Design

To make an instance of a memory or constant runtime-modifiable:

1. Open the instance with the Parameter Editor.
2. In the Parameter Editor, turn on **Allow In-System Memory Content Editor to capture and update content independently of the system clock**.



System: RAM **Path:** ram_1port_0

RAM: 1-PORT
ram_1port

Memory Initial Mode

Do you want to specify the initial content of the memory?: ● No, leave it blank
○ Yes, use this file for the memory content data

□ Initialize memory content data to XX..X on power-up in simulation

(You can use a Hexadecimal (Intel-format) File [.hex] or a Memory Initialization File [.mif])

File name:

In-System Memory Content Editor

□ Allow In-System Memory Content Editor to capture and update content independently of the system clock

The 'instance ID' of this RAM is:

3. Recompile the design.

When you specify that a memory or constant is run-time modifiable, the Intel Quartus Prime software changes the default implementation to enable run-time modification without changing the functionality of your design, by:

- Converting single-port RAMs to dual-port RAMs
- Adding logic to avoid memory write collision and maintain read write coherency in device families that do not support true dual-port RAMs, such as Intel Stratix 10.

5.4. Programming the Device with the In-System Memory Content Editor

After compilation, you must program the design in the FPGA. You can use the JTAG Chain Configuration Pane to program the device from within the In-System Memory Content Editor.

Related Information

JTAG Chain Configuration Pane (In-System Memory Content Editor)
In *Intel Quartus Prime Help*

5.5. Loading Memory Instances to the ISMCE

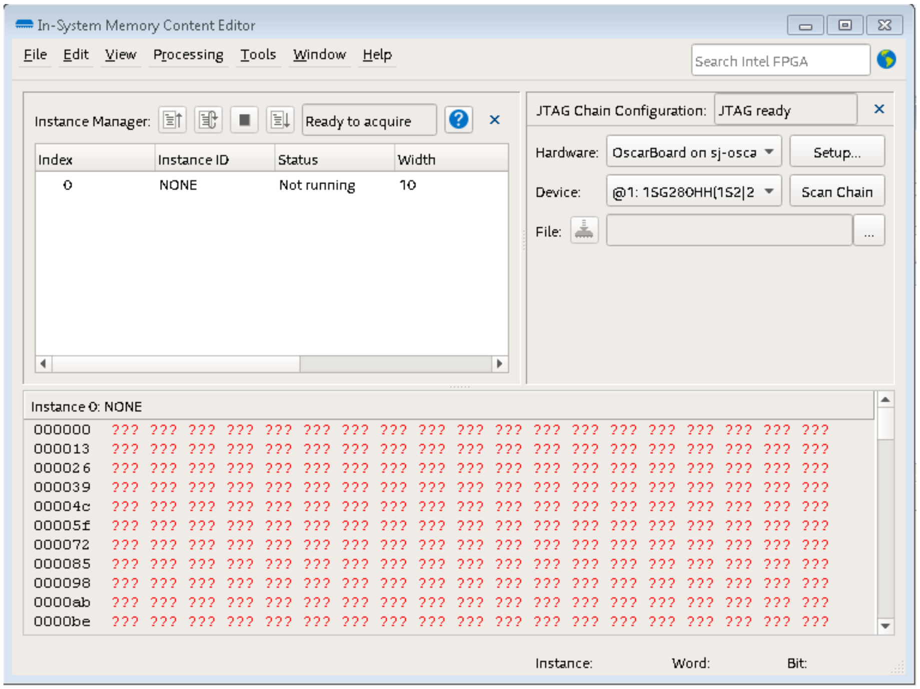
To view the content of reconfigurable memory instances:

1. On the Intel Quartus Prime software, click **Tools ➤ In-System Memory Content Editor**.
2. In the **JTAG Chain Configuration** pane, click **Scan Chain**.

   The In-System Memory Content Editor sends a query to the device in the **JTAG Chain Configuration** pane and retrieves all instances of run-time configurable memories and constants.

   The **Instance Manager** pane lists all the instances of constants and memories that are runtime-modifiable. The **Hex Editor** pane displays the contents of each memory or constant instance. The memory contents in the **Hex Editor** pane appear as red question marks until you read the device.

Figure 102. Hex Editor After Scanning JTAG Chain



Instance Manager: Ready to acquire

Index	Instance ID	Status	Width
0	NONE	Not running	10

JTAG Chain Configuration: JTAG ready

Hardware: OscarBoard on sj-osca

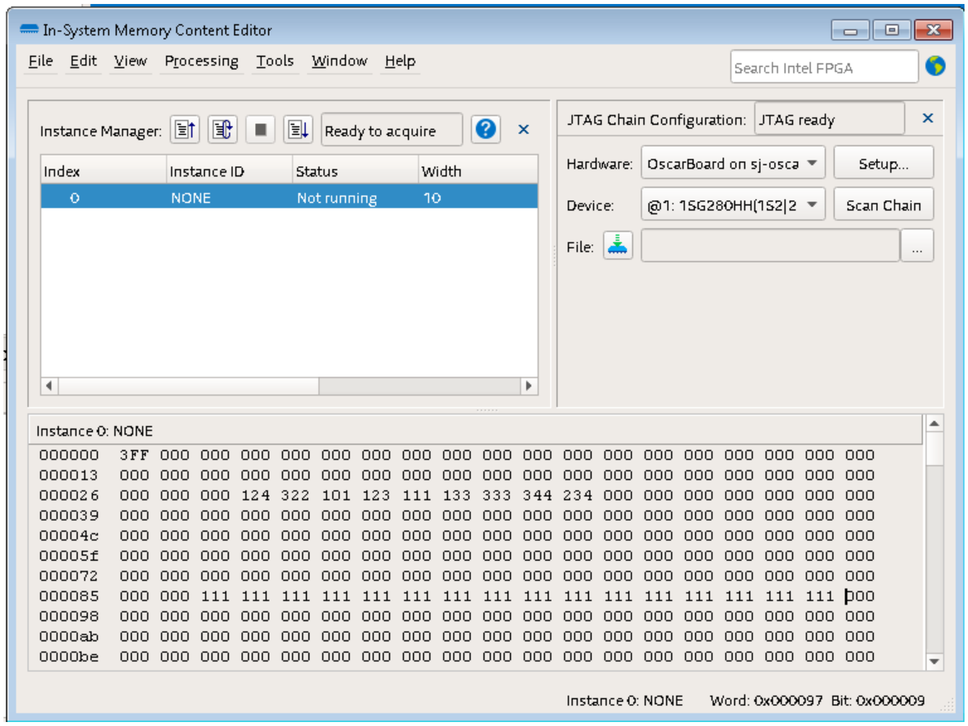
Device: @1: 1SG280HH(1S2|2

Instance 0: NONE

```
000000  ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ???
000013  ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ???
000026  ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ???
000039  ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ???
00004c  ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ???
00005f  ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ???
000072  ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ???
000085  ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ???
000098  ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ???
0000ab  ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ???
0000be  ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ??? ???
```

3. Click an instance from the **Instance manager**, and then click ↑ to load the contents of that instance.

The Hex Editor now displays the contents of the instance.



Instance Manager: Ready to acquire

Index	Instance ID	Status	Width
0	NONE	Not running	10

JTAG Chain Configuration: JTAG ready

Hardware: OscarBoard on sj-osca

Device: @1: 1SG280HH(1S2|2

File:

Instance 0: NONE

```
000000   3FF 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000013   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000026   000 000 000 124 322 101 123 111 133 333 344 234 000 000 000 000 000 000 000
000039   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
00004c   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
00005f   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000072   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000085   000 000 111 111 111 111 111 111 111 111 111 111 111 111 111 111 111 111 000
000098   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0000ab   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0000be   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
```

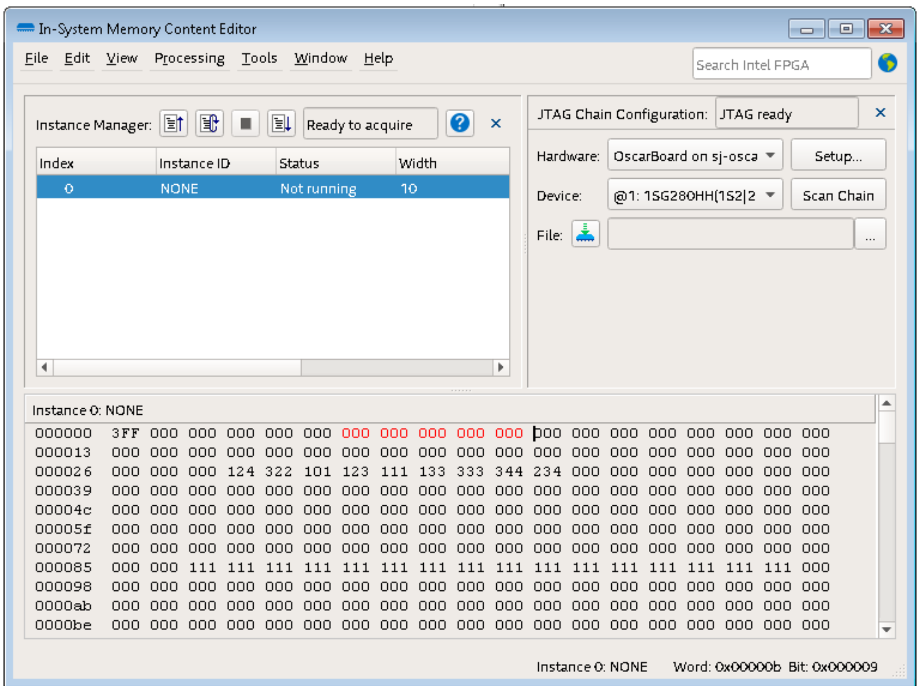
Instance 0: NONE   Word: 0x000097 Bit: 0x000009

5.6. Monitoring Locations in Memory

The ISMCE allows you to monitor information in memory regions. For example, you can determine if a counter increments, or if a given word changes. For memories connected to a NIOS processor, you can observe how the software uses key regions of memory.

- Click ⬆ to synchronize the Hex Editor to the current instance's content.
  The Hex Editor displays in red content that changed with respect to the last device synchronization.

Figure 103. Hex Editor after Manually Editing Content



Instance Manager: Ready to acquire

Index	Instance ID	Status	Width
0	NONE	Not running	10

JTAG Chain Configuration: JTAG ready

Hardware: OscarBoard on sj-osca

Device: @1: 1SG280HH(1S2|2

Instance 0: NONE

```
000000  3FF 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000013  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000026  000 000 000 124 322 101 123 111 133 333 344 234 000 000 000 000 000 000 000
000039  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
00004c  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
00005f  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000072  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000085  000 000 111 111 111 111 111 111 111 111 111 111 111 111 111 111 111 111 000
000098  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0000ab  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0000be  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
```

Instance 0: NONE   Word: 0x00000b Bit: 0x000009

- If you want a live output of the memory contents instead of manually synchronizing, click ⟳.

  Continuous read is analogous to using Signal Tap in continuous acquisition, with the memory values appearing as words in the Hex Editor instead of toggling waveforms.

Note: (Intel Stratix 10 only) ISMCE logic can perform Read/Write operations only when the design logic is idle. If the design logic attempts a write or an address change operation, the design logic prevails, and the ISMCE operation times out. An error message lets you know that the memory connected to the In-System Memory Content Editor instance is in use, and memory content is not updated.

Related Information

- Read Information from In-System Memory Commands (Processing Menu)
  In *Intel Quartus Prime Help*
- Stop In-System Memory Analysis Command (Processing Menu)
  In *Intel Quartus Prime Help*

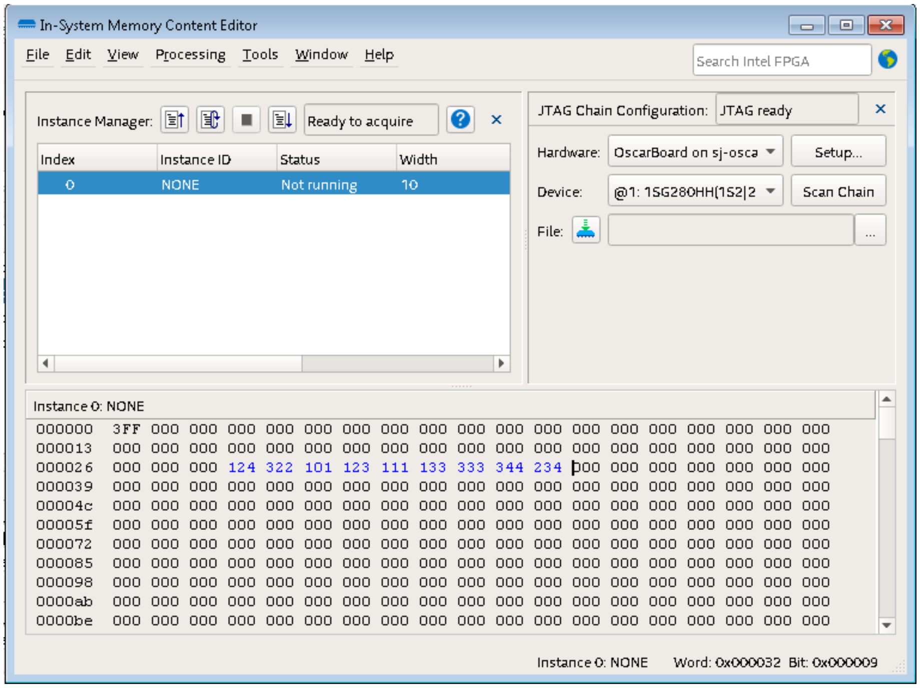
5.7. Editing Memory Contents with the Hex Editor Pane

You can edit the contents of instances by typing values directly into the **Hex Editor** pane.

Black content on the **Hex Editor** pane means that the value read is the same as last synchronization.

1. Type content on the pane.
   The **Hex Editor** displays in blue changed content that has not been synchronized to the device.

Figure 104. Hex Editor after Manually Editing Content



In-System Memory Content Editor
File Edit View Processing Tools Window Help
Search Intel FPGA
Instance Manager:
Setup...
Scan Chain
...

Ready to acquire

Index	Instance ID	Status	Width
0	NONE	Not running	10

JTAG Chain Configuration: JTAG ready

Hardware: OscarBoard on sj-osca

Device: @1: 1SG280HH(1S2|2

File:

Instance 0: NONE

```
000000  3FF 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000013  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000026  000 000 000 124 322 101 123 111 133 333 344 234 000 000 000 000 000 000 000
000039  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
00004c  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
00005f  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000072  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000085  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
000098  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0000ab  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0000be  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
```

Instance 0: NONE   Word: 0x000032 Bit: 0x000009

2. Click to synchronize the content to the device.

Note: (Intel Stratix 10 only) ISMCE logic can perform Read/Write operations only when the design logic is idle. If the design logic attempts a write or an address change operation, the design logic prevails, and the ISMCE operation times out. An error message lets you know that the memory connected to the In-System Memory Content Editor instance is in use, and reports the number of successful writes before the design logic requested access to the memory.

Related Information

- Custom Fill Dialog Box
  In *Intel Quartus Prime Help*
- Write Information to In-System Memory Commands (Processing Menu)
  In *Intel Quartus Prime Help*
- Go To Dialog Box
  In *Intel Quartus Prime Help*
- Select Range Dialog Box
  In *Intel Quartus Prime Help*

5.8. Importing and Exporting Memory Files

The In-System Memory Content Editor allows you to import and export data values for memories that are runtime modifiable. Importing from a data file enables you to quickly load an entire memory image. Exporting to a data file allows you to save the contents of the memory for future use.

You can import or export files in `hex` or `mif` formats.

1. To import a file, click **Edit ➤ Import Data from File...**, and then select the file to import.

   If the file is not compatible, unexpected data appears in the Hex Editor.

2. To export memory contents to a file, click **Edit ➤ Export Data to File...**, and then specify the name.

Related Information

- Import Data
  In *Intel Quartus Prime Help*
- Export Data
  In *Intel Quartus Prime Help*
- Hexadecimal (Intel-Format) File (.hex) Definition
  In *Intel Quartus Prime Help*
- Memory Initialization File (.mif) Definition
  In *Intel Quartus Prime Help*

5.9. Access Two or More Devices

If you have more than one device with in-system configurable memories or constants in a JTAG chain, you can launch multiple In-System Memory Content Editors within the Intel Quartus Prime software to access the memories and constants in each of the devices. Each window of the In-System Memory Content Editor can access the memories and constants of a single device.

5.10. Scripting Support

The Intel Quartus Prime software allows you to perform runtime modification of memories and constants in scripted flows.

You can enable memory and constant instances to be runtime modifiable from the HDL code. Additionally, the In-System Memory Content Editor supports reading and writing of memory contents via Tcl commands from the `insystem_memory_edit` package.

Related Information

- Tcl Scripting
  In *Intel Quartus Prime Pro Edition User Guide: Scripting*
- Command Line Scripting
  In *Intel Quartus Prime Pro Edition User Guide: Scripting*

5.10.1. The insystem_memory_edit Tcl Package

The **::quartus::insystem_memory_edit** Tcl package contains the set of Tcl functions for reading and editing the contents of memory in an Intel FPGA device using the In-System Memory Content Editor. The `quartus_stp` and `quartus_stp_tcl` command line executables load this package by default.

For the most up-to-date information about the **::quartus::insystem_memory_edit**, refer to the Intel Quartus Prime Help.

Related Information

::quartus::insystem_memory_edit
In *Intel Quartus Prime Help*

5.10.1.1. Getting Information about the insystem_memory_edit Package

You can also get information on the `insystem_memory_edit` package directly from the command line:

- For general information about the package, type:

```
quartus_stp --tcl_eval help -pkg insystem_memory_edit
```

- For information about a command in the package, type:

```
quartus_stp --tcl_eval help -cmd <command_name>
```

5.11. In-System Modification of Memory and Constants Revision History

The following revision history applies to this chapter:

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0.0	• Added support for the Intel Stratix 10 device family. • Removed obsolete example.
2016.10.31	16.1.0	• Implemented Intel rebranding.
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
June 2014	14.0.0	• Dita conversion. • Removed references to megafunction and replaced with IP core.
June 2012	12.0.0	Removed survey link.
November 2011	10.0.3	Template update.
December 2010	10.0.2	Changed to new document template. No change to content.
August 2010	10.0.1	Corrected links
July 2010	10.0.0	• Inserted links to Intel Quartus Prime Help • Removed Reference Documents section
November 2009	9.1.0	• Delete references to APEX devices • Style changes

continued...

Document Version	Intel Quartus Prime Version	Changes
March 2009	9.0.0	No change to content
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	8.0.0	• Added reference to Section V. In-System Debugging in volume 3 of the Intel Quartus Prime Handbook on page 16-1 • Removed references to the Mercury device, as it is now considered to be a "Mature" device • Added links to referenced documents throughout document • Minor editorial updates

6. Design Debugging Using In-System Sources and Probes

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

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

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

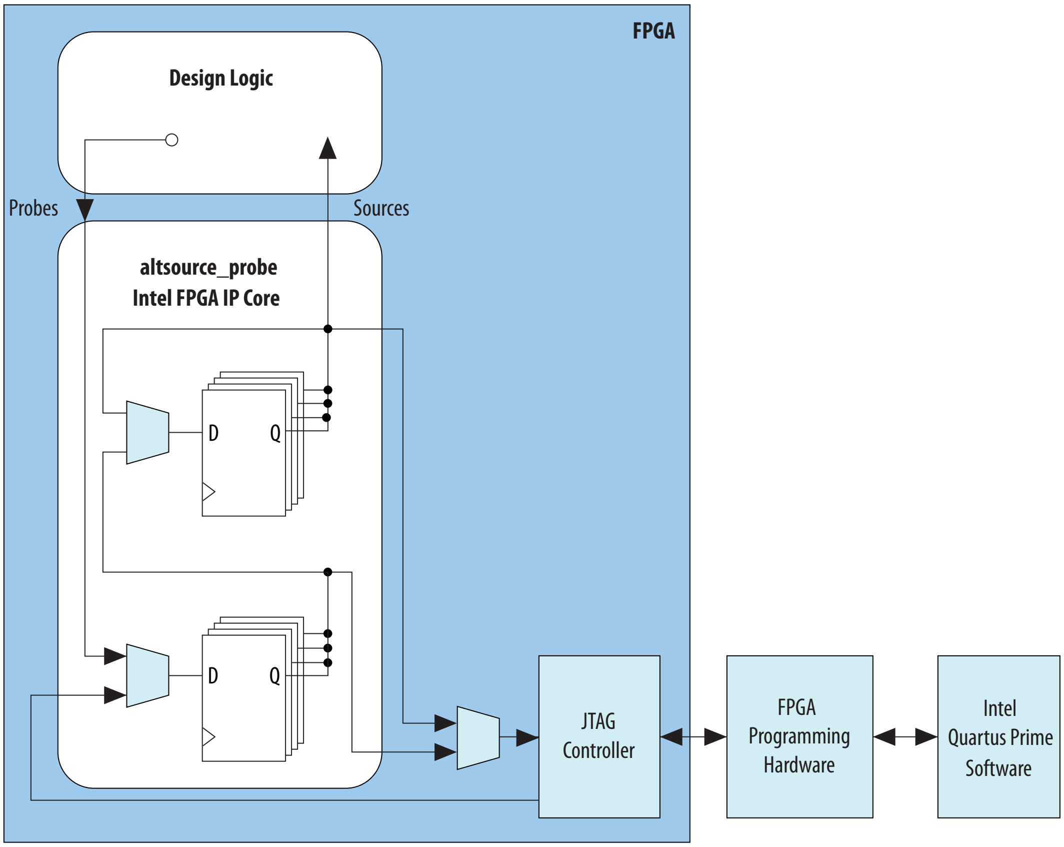
- Force the occurrence of trigger conditions set up in the Signal Tap Logic Analyzer
- Create simple test vectors to exercise your design without using external test equipment
- Dynamically control run time control signals with the JTAG chain

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

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

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

Figure 105. In-System Sources and Probes Editor Block Diagram



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

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

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

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

Related Information

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

6.1. Hardware and Software Requirements

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

- Intel Quartus Prime software

or

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

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

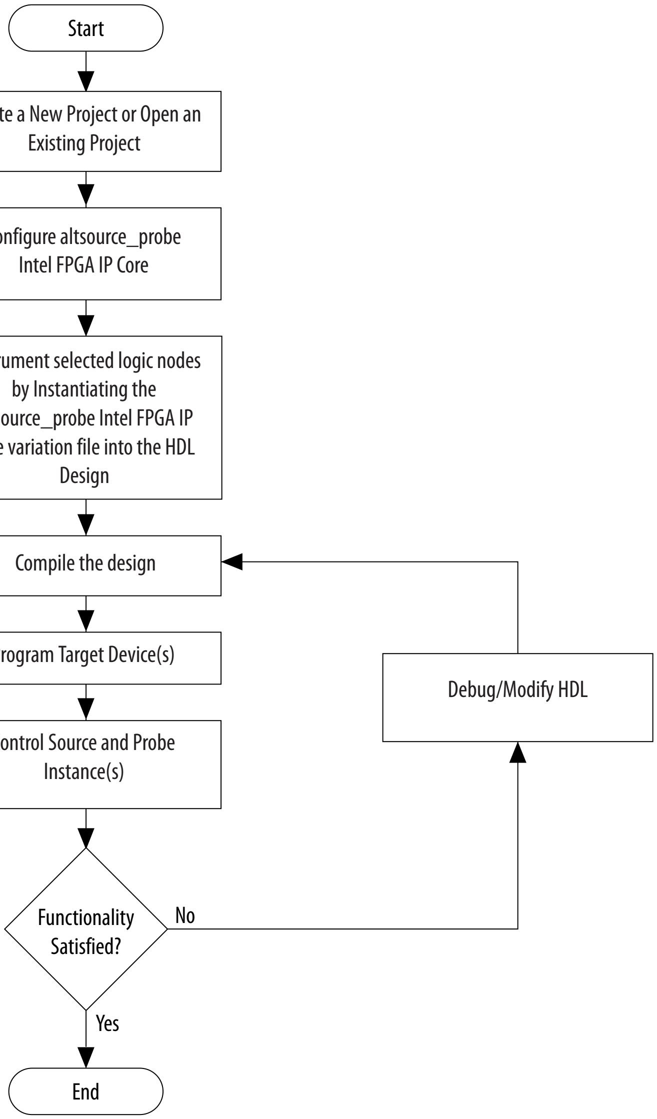
- Arria series
- Stratix series
- Cyclone® series
- MAX® series

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

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

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

Figure 106. FPGA Design Flow Using the In-System Sources and Probes Editor


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

6.2.1. Instantiating the In-System Sources and Probes IP Core

To instantiate the In-System Sources and Probes IP core in a design:

1. In the IP Catalog (**Tools ➤ IP Catalog**), type `In-System Sources and Probes`.
2. Double-click **In-System Sources and Probes** to open the parameter editor.
3. Specify a name for the IP variation.
4. Specify the parameters for the IP variation.

The IP core supports up to 512 bits for each source, and design can 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.
6. Using the generated template, instantiate the In-System Sources and Probes IP core in your design.

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

6.2.2. In-System Sources and Probes IP Core Parameters

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

Table 33. 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.

6.3. Compiling the Design

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.

6.4. Running the In-System Sources and Probes Editor

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

To run the In-System Sources and Probes Editor:

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

6.4.1. In-System Sources and Probes Editor GUI

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

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

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

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

6.4.2. Programming Your Device With JTAG Chain Configuration

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

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

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

6.4.3. Instance Manager

The **Instance Manager** pane provides a list of all ALTSOURCE_PROBE instances in the design, and allows you to configure data acquisition.

The **Instance Manager** pane contains the following buttons and sub-panes:

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

Beside each entry, the **Instance Manager** pane displays the instance status. The possible instance statuses are **Not running Offloading data**, **Updating data**, and **Unexpected JTAG communication error**.

6.4.4. In-System Sources and Probes Editor Pane

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

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

6.4.4.1. Reading Probe Data

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

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

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

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

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

6.4.4.2. Writing Data

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

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

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

6.4.4.3. Organizing Data

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

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

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

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

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

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

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

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

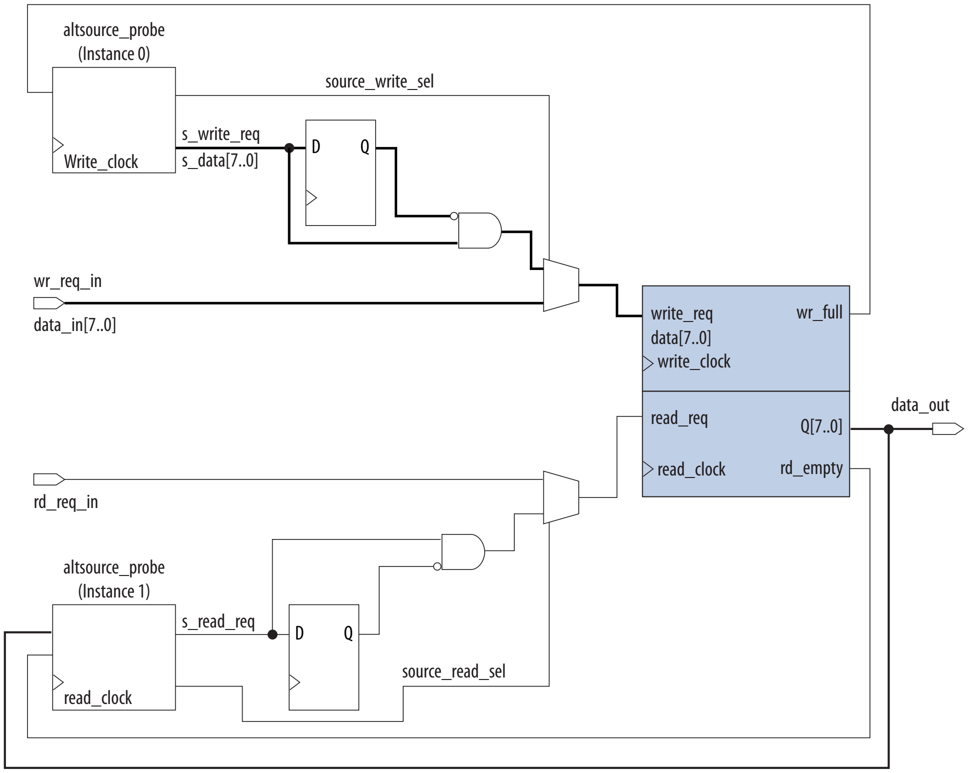
Table 34. In-System Sources and Probes Tcl Commands

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

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

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

Figure 107. 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

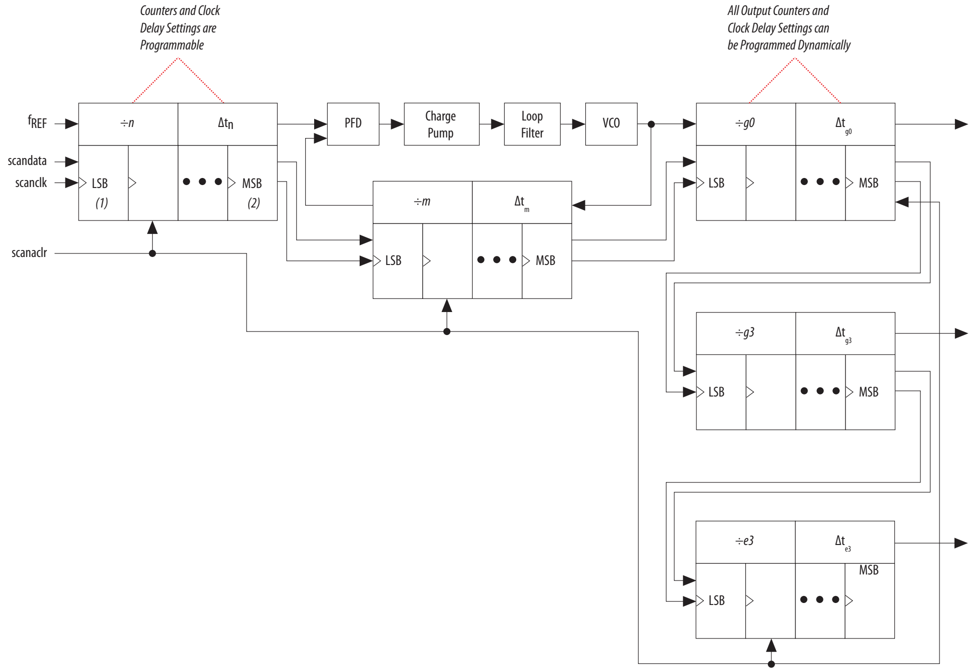
- Tcl Scripting
- Intel Quartus Prime Settings File Manual
- Command Line Scripting

6.6. Design Example: Dynamic PLL Reconfiguration

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

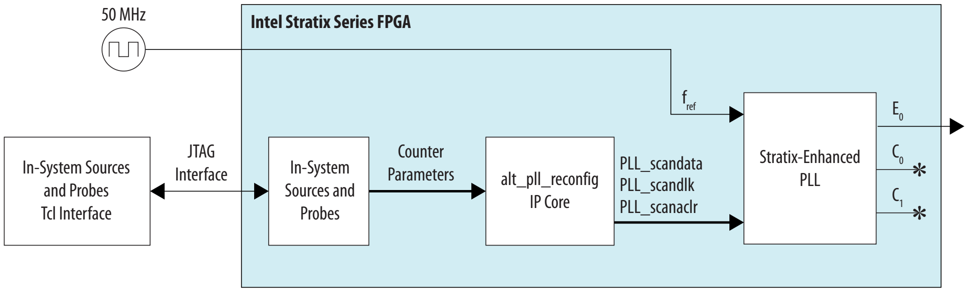
Stratix PLLs allow you to dynamically update PLL coefficients during run time. Each enhanced PLL within the Stratix device contains a register chain that allows you to modify the pre-scale counters (m and n values), output divide counters, and delay counters. In addition, the ALTPLL_RECONFIG IP core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

Figure 108. 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
MSB
LSB

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.

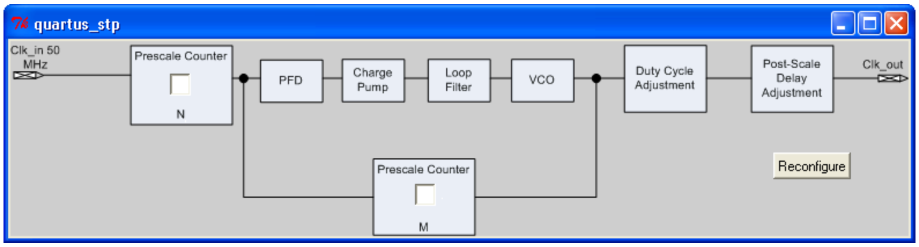
Figure 109. Block Diagram of Dynamic PLL Reconfiguration Design Example



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

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

Figure 110. 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

6.7. Design Debugging Using In-System Sources and Probes Revision History

The following revision history applies to this chapter:

Document Version	Intel Quartus Prime Version	Changes
2019.06.11	18.1.0	Rebranded megafunction to Intel FPGA IP core
2018.05.07	18.0.0	Added details on finding the In-System Sources and Probes in the IP Catalog.
2016.10.31	16.1.0	Implemented Intel rebranding.
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
June 2014	14.0.0	Updated formatting.
June 2012	12.0.0	Removed survey link.
November 2011	10.1.1	Template update.
December 2010	10.1.0	Minor corrections. Changed to new document template.
July 2010	10.0.0	Minor corrections.
November 2009	9.1.0	• Removed references to obsolete devices. • Style changes.

continued...

Document Version	Intel Quartus Prime Version	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

7. Analyzing and Debugging Designs with System Console

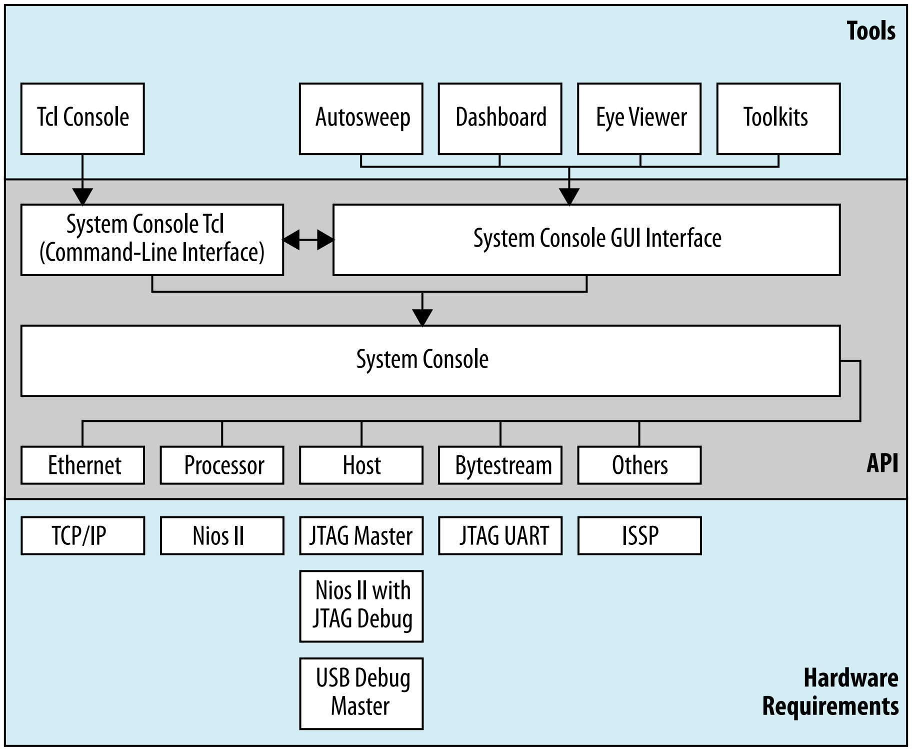
7.1. Introduction to System Console

System Console provides visibility into your design and allows you to perform system-level debug on an FPGA at run-time. System Console performs tests on debug-enabled Intel FPGA IP. A variety of debug services provide read and write access to elements in your design.

- Perform board bring-up with finalized or partially complete designs.
- Automate run-time verification through scripting across multiple devices.
- Debug transceiver links, memory interfaces, and Ethernet interfaces.
- Integrate your debug IP into the debug platform.
- Perform system verification with MATLAB and Simulink.

Figure 111. System Console Tools

The System Console API supports services that access your design in operation.



System Console also provides the hardware debugging infrastructure to support operation and customization of debugging "toolkits." Toolkits are small applications that you can use to perform system-level debug of such elements as external memory interfaces, Ethernet interfaces, PCI Express interfaces, transceiver PHY interfaces, and various other debugging functions. The Intel Quartus Prime software includes the Available System Debugging Toolkits on page 174. For advanced users, System Console also supports Tcl commands that allow you to define and operate your own custom toolkits, as *System Console and Toolkit Tcl Command Reference Manual* describes.

Related Information

- System Console Online Training
- System Console and Toolkit Tcl Command Reference Manual

7.1.1. IP Cores that Interact with System Console

System Console runs on your host computer and communicates with your running design through debug agents. Debug agents are the soft-logic embedded in some IP cores that enable debug communication with the host computer.

You can instantiate debug IP cores using the Intel Quartus Prime software IP Catalog and IP parameter editor. Some IP cores are enabled for debug by default, while you must enable debug for other IP cores through options in the parameter editor. Some debug agents have multiple purposes.

When you include debug-enabled IP cores in your design, you can access large portions of the design running on hardware for debugging purposes. Debug agents allow you to read and write to memory and alter peripheral registers from the tool.

Services associated with debug agents in the running design can open and close as needed. System Console determines the communication protocol with the debug agent. The communication protocol determines the best board connection to use for command and data transmission.

The Programmable SRAM Object File (`.sof`) that the Intel Quartus Prime Assembler generates for device programming provides the System Console with channel communication information. When you open System Console from the Intel Quartus Prime software GUI, with a project open that includes a `.sof`, System Console automatically finds and links to the device(s) it detects. When you open System Console without an open project, or with an unrelated project open, you can manually load the `.sof` file that you want, and then the design linking occurs automatically if the device(s) match.

Related Information

- Available System Debugging Toolkits on page 174
- WP-01170 System-Level Debugging and Monitoring of FPGA Designs

7.1.2. Services Provided through Debug Agents

By adding the appropriate debug agent to your design, System Console services can use the associated capabilities of the debug agent.

Table 35. Common Services for System Console

Service	Function	Debug Agent Providing Service
host	Access a memory-mapped agent connected to the host interface.	• Nios II with debug • JTAG to Avalon Master Bridge • USB Debug Master
agent	Allows a host component to access a single agent without needing to know the location of the agent in the host's memory map. Any agent that is accessible to a System Console host can provide this service.	• Nios II with debug • JTAG to Avalon Master Bridge • USB Debug Master If an SRAM Object File (`.sof`) is loaded, then agents accessed by a debug host provide the agent service.
issp	The In-System Sources and Probes (ISSP) service provides scriptable access to the In-System Sources and Probes Intel FPGA IP for generating stimuli and soliciting responses from your logic design.	In-System Sources and Probes Intel FPGA IP
processor	• Start, stop, or step the processor. • Read and write processor registers.	Nios II with debug
JTAG UART	The JTAG UART is an Avalon memory mapped agent that you can use in conjunction with System Console to send and receive byte streams.	JTAG UART

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

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

Related Information

Available System Debugging Toolkits on page 174

7.1.3. System Console Debugging Flow

The System Console debugging flow includes the following steps:

1. Add debug-enabled Intel FPGA IP to your design.
2. Compile the design.
3. Connect to a board and program the FPGA.
4. Start System Console.
5. Locate and open a System Console service.
6. Perform debug operations with the service.
7. Close the service.

Related Information

- Starting System Console on page 161
- Launching a Toolkit in System Console on page 172
- Using System Console Services on page 176

- Running System Console in Command-Line Mode on page 191

7.2. Starting System Console

You can use any of the following methods to start System Console:

- To start System Console from the Intel Quartus Prime software GUI:

  Click **Tools ➤ System Debugging Tools ➤ System Console**.

  Or

  Click **Tools ➤ System Debugging Tools ➤ System Debugging Toolkits**.
- To start System Console from Platform Designer:

  Click **Tools ➤ System Console**
- To start Stand-Alone System Console:
  1. Navigate to the **Download Center** page, click **Additional Software**, and download and install Intel Quartus Prime Pro Edition Programmer and Tools.
  2. On the Windows Start menu, click **All Programs ➤ Intel FPGA <*version*> ➤ Programmer and Tools ➤ System Console**.
- To start System Console from a Nios II Command Shell:
  1. On the Windows Start menu, click **All Programs ➤ Intel ➤ Nios II EDS <*version*> ➤ Nios II<*version*> ➤ Command Shell**.
  2. Type `system-console --project_dir=<project directory>` and specify a directory that contains `.qsf` or `.sof` files.

     Note: Type `--help` for System Console help.

Related Information

Download Center for FPGAs

7.2.1. Customizing System Console Startup

You can customize your System Console startup environment, as follows:

- Add commands to the `system_console_rc` configuration file located at:
  - `<$HOME>/system_console/system_console_rc.tcl`

  The file in this location is the user configuration file, which only affects the owner of the home directory.
- Specify your own design startup configuration file with the command-line argument `--rc_script=<path_to_script>`, when you launch System Console from the Nios II command shell.
- Use the `system_console_rc.tcl` file in combination with your custom `rc_script.tcl` file. In this case, the `system_console_rc.tcl` file performs System Console actions, and the `rc_script.tcl` file performs your debugging actions.

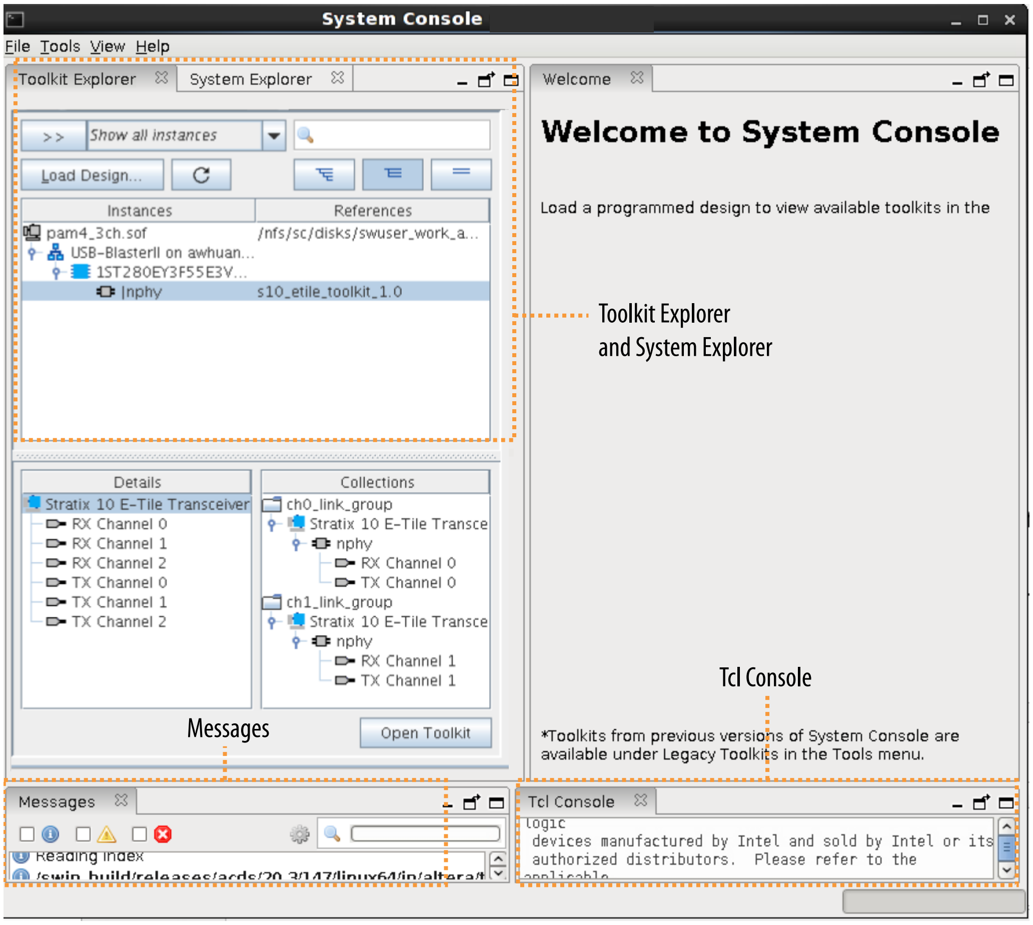
On startup, System Console automatically runs the Tcl commands in these files. The commands in the `system_console_rc.tcl` file run first, followed by the commands in the `rc_script.tcl` file.

7.3. System Console GUI

The System Console GUI consists of a main window with the following panes that allow you to interact with the design currently running on the FPGA device:

- **Toolkit Explorer**—displays all available toolkits and launches tools that use the System Console framework.
- **System Explorer**—displays a list of interactive instances in your design, including board connections, devices, designs, servers, and scripts.
- **Main View**—initially displays the welcome screen. All toolkits that you launch display in this view.
- **Tcl Console**—allows you to interact with your design through individual Tcl commands or by sourcing Tcl scripts, writing procedures, and using System Console APIs.
- **Messages**—displays status, warning, and error messages related to connections and debug actions.

Figure 112. System Console GUI



Welcome to System Console

Load a programmed design to view available toolkits in the

*Toolkits from previous versions of System Console are available under Legacy Toolkits in the Tools menu.

```
Reading index
/swin_build/releases/acds/20.3/147/linux64/ip/altera/t
```

```
logic
 devices manufactured by Intel and sold by Intel or its
 authorized distributors.  Please refer to the
applicable
```

System Console GUI also provides the **Autosweep**, **Dashboard**, and **Eye Viewer** panes, that display as tabs in the **Main View**.

System Console Views on page 163
Toolkit Explorer Pane on page 171
System Explorer Pane on page 171

Related Information

System Console and Toolkit Tcl Command Reference Manual

7.3.1. System Console Views

System Console provides the following views:

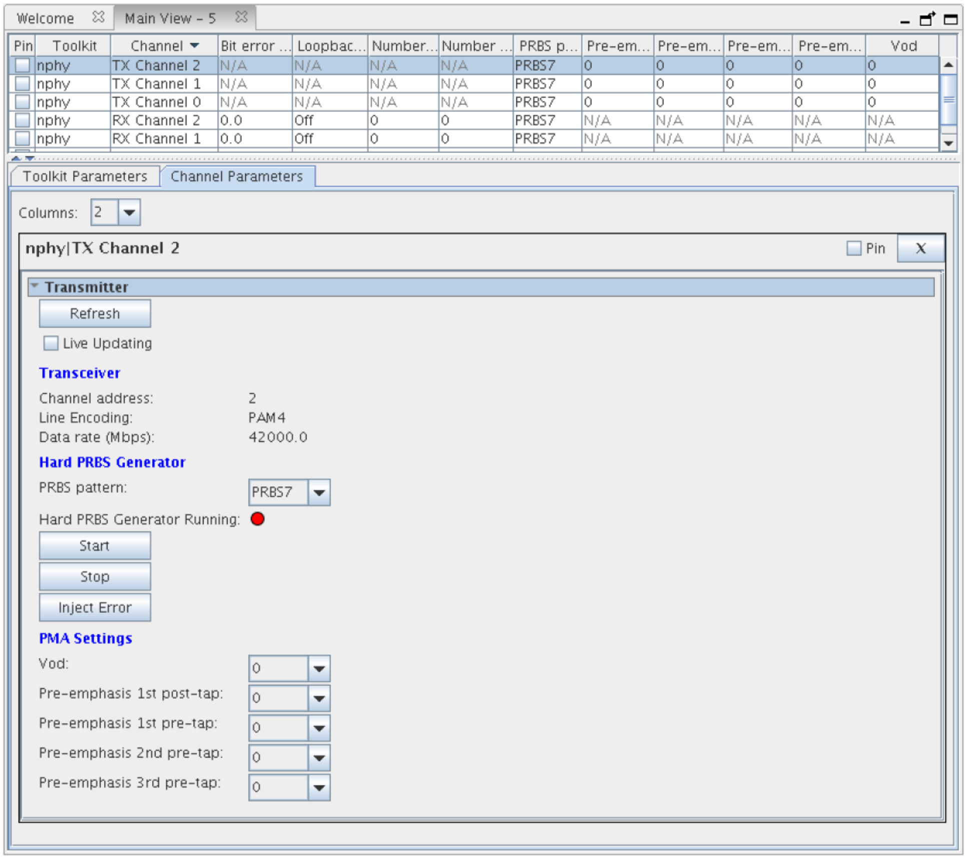
Main View on page 164
Autosweep View on page 167
Dashboard View on page 168
Eye Viewer on page 169

7.3.1.1. Main View

The **Main View** in System Console allows you to visualize certain parameter values of the IP that the toolkit targets. These parameter values can be static values from compile-time parameterization, or dynamic values read from the hardware (like reading from CSR registers) at run-time.

The **Main View** GUI controls allow you to control or configure the IP on the hardware.

Figure 113. Main View of the System Console



Pin	Toolkit	Channel ▾	Bit error ...	Loopbac...	Number...	Number ...	PRBS p...	Pre-em...	Pre-em...	Pre-em...	Pre-em...	Vod
□	nphy	TX Channel 2	N/A	N/A	N/A	N/A	PRBS7	0	0	0	0	0
□	nphy	TX Channel 1	N/A	N/A	N/A	N/A	PRBS7	0	0	0	0	0
□	nphy	TX Channel 0	N/A	N/A	N/A	N/A	PRBS7	0	0	0	0	0
□	nphy	RX Channel 2	0.0	Off	0	0	PRBS7	N/A	N/A	N/A	N/A	N/A
□	nphy	RX Channel 1	0.0	Off	0	0	PRBS7	N/A	N/A	N/A	N/A	N/A

Columns: 2

nphy|TX Channel 2 □ Pin X

□ Live Updating

Transceiver

Channel address: 2
Line Encoding: PAM4
Data rate (Mbps): 42000.0

Hard PRBS Generator

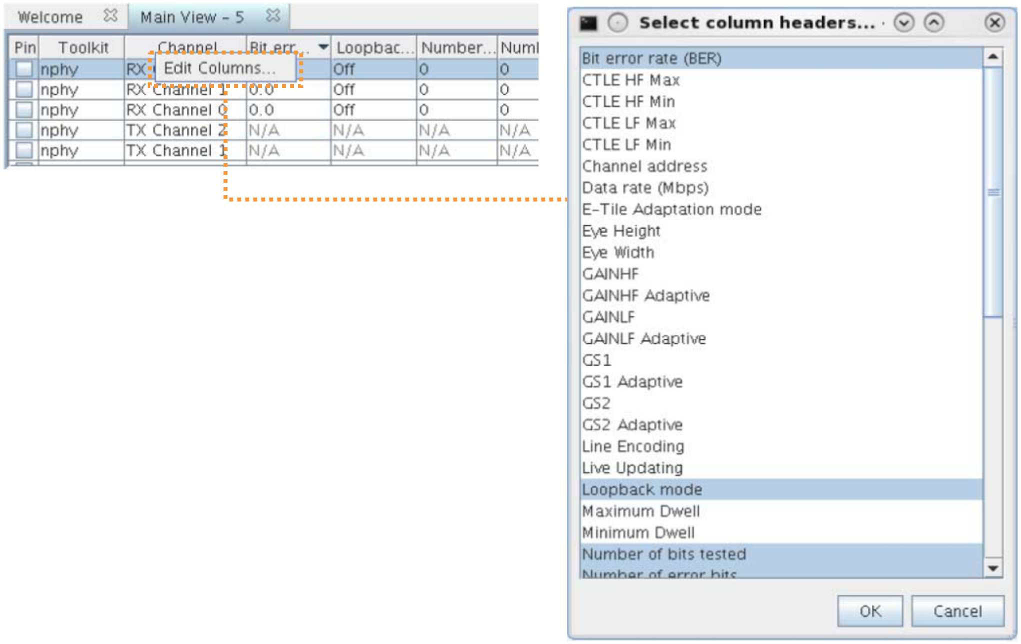
PRBS pattern: PRBS7
Hard PRBS Generator Running: ●

PMA Settings

Vod: 0
Pre-emphasis 1st post-tap: 0
Pre-emphasis 1st pre-tap: 0
Pre-emphasis 2nd pre-tap: 0
Pre-emphasis 3rd pre-tap: 0

If you are using a toolkit, you can add or remove columns from the table in the **Main View**. Right-click on the table header and select **Edit Columns** in the right-click menu. The **Select column headers** dialog box is displayed where you can choose to include more columns, as shown in the following image:

Figure 114. Column Selection in the Main View Table



Pin	Toolkit	Channel	Bit err...	Loopbac...	Number...	Numl
□	nphy	RX		Off	0	0
□	nphy	RX Channel 1	0.0	Off	0	0
□	nphy	RX Channel 0	0.0	Off	0	0
□	nphy	TX Channel 2	N/A	N/A	N/A	N/A
□	nphy	TX Channel 1	N/A	N/A	N/A	N/A

Bit error rate (BER)
CTLE HF Max
CTLE HF Min
CTLE LF Max
CTLE LF Min
Channel address
Data rate (Mbps)
E-Tile Adaptation mode
Eye Height
Eye Width
GAINHF
GAINHF Adaptive
GAINLF
GAINLF Adaptive
GS1
GS1 Adaptive
GS2
GS2 Adaptive
Line Encoding
Live Updating
Loopback mode
Maximum Dwell
Minimum Dwell
Number of bits tested
Number of error bits

Parameters Pane

The **Main View** provides the **Parameters** pane that has two tabs, one for global parameters (those not associated with a given channel) and another for channel parameters (those associated with channels). The **Channel Parameters** tab is filled with per-channel parameter editors based on channel row selection in the **Status Table**, as Figure 120 on page 171 shows.

Status Table Pane

The **Status Table** does not appear for toolkits that do not define channels. The **Status Table** allows you to view status information across all channels of a collection or a toolkit instance, as well as execute actions across multiple channels, as Figure 120 on page 171 shows. You can execute bulk actions spanning multiple channels by selecting desired channels, and right-clicking and exploring the **Actions** sub-menu.

You can also use the **Status Table** to select which channel to display in the Parameters Pane on page 165. The channels you select in the Status Table are shown in the **Parameters Pane**. You can use the **Pin** setting for a channel to display the channel, regardless of the current selection in the **Status Table**.

If you develop your own toolkit, you can design the layout and GUI elements in the **Main View** using the Toolkit Tcl API. You can also define how each GUI element interacts with the hardware.

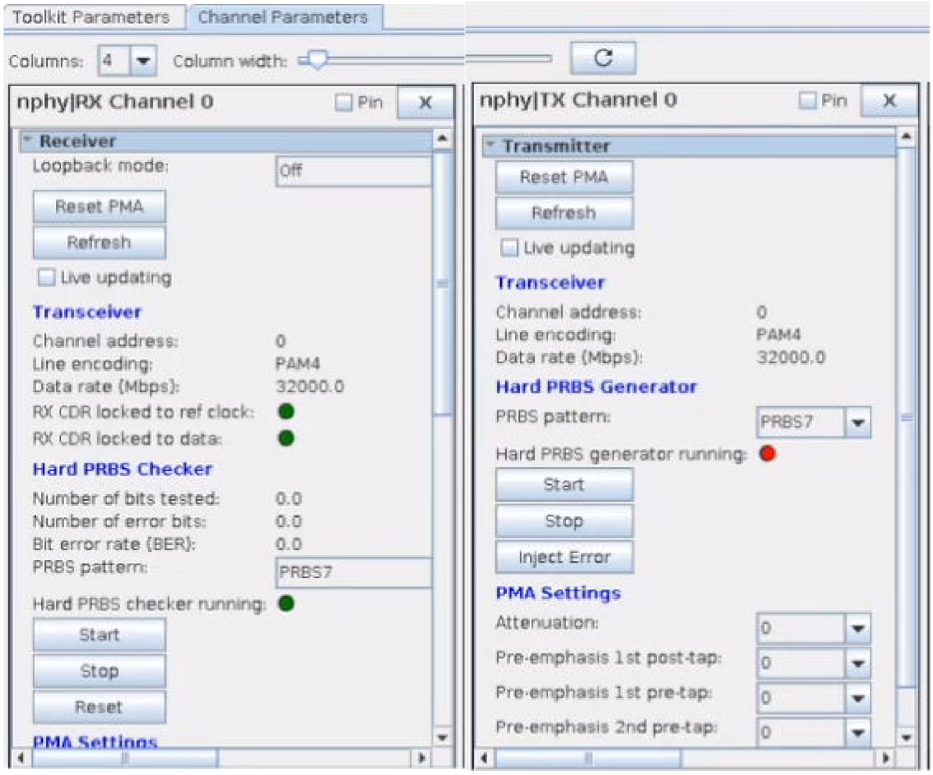
7.3.1.1.1. Link Pair View

Displaying Links With the Main View

You can use the **Main View** to simultaneously display and control associated TX and RX pairs:

1. Select both RX and TX channels in the **Status Table**.
2. Right-click to view the context-sensitive menu.
3. Navigate to the **Actions** menu.

Figure 115. Displaying Links in the Main View



Columns: 4

nphy|RX Channel 0

Receiver

Loopback mode: Off

□ Live updating

Transceiver

Channel address: 0
Line encoding: PAM4
Data rate (Mbps): 32000.0
RX CDR locked to ref clock: ●
RX CDR locked to data: ●

Hard PRBS Checker

Number of bits tested: 0.0
Number of error bits: 0.0
Bit error rate (BER): 0.0
PRBS pattern: PRBS7
Hard PRBS checker running: ●

nphy|TX Channel 0

Transmitter

□ Live updating

Transceiver

Channel address: 0
Line encoding: PAM4
Data rate (Mbps): 32000.0

Hard PRBS Generator

PRBS pattern: PRBS7
Hard PRBS generator running: ●

PMA Settings

Attenuation: 0
Pre-emphasis 1st post-tap: 0
Pre-emphasis 1st pre-tap: 0
Pre-emphasis 2nd pre-tap: 0

Custom Groups with Links

In the **Status Table**, links are displayed like any other channel, with the exception that their parameter lists encompass all parameters from the associated TX and RX channels. If you create a group with a link and its associated TX and RX instance channels, the link row in the **Status Table** populates in all columns. Whereas, for the independent TX and RX channel rows, only parameters associated with that channel populate the **Status Table**.

Configuring a Link

You have the option to configure links in the following ways:

- Through the provided status table in the **Main View**.
- Through configuration options provided for the associated TX and RX channels.

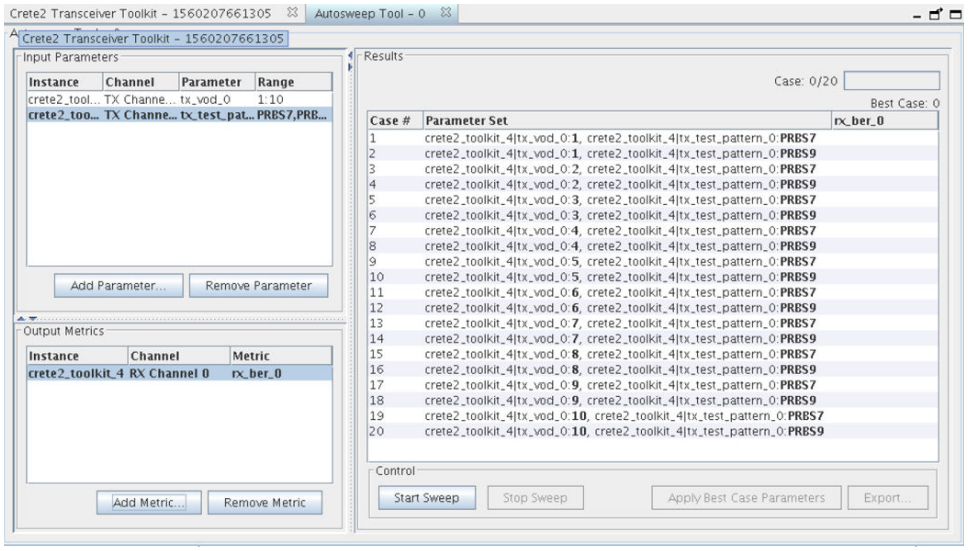
Option provided for the TX and RX channels allow you to individually manipulate each associated channel. You can manipulate multiple parameters simultaneously by selecting one or more items, and then right-clicking parameters in the status table.

7.3.1.2. Autosweep View

The **Autosweep** view allows you to sweep over IP parameters to find the best combination and define your own quality metrics for a given Autosweep run.

The System Console **Autosweep** view allows you to define your own quality metrics for a given Autosweep run.

Figure 116. Autosweep View of the System Console



Instance	Channel	Parameter	Range
crete2_tool...	TX Channe...	tx_vod_0	1:10
crete2_too...	TX Channe...	tx_test_pat...	PRBS7,PRB...

Instance	Channel	Metric
crete2_toolkit_4	RX Channel 0	rx_ber_0

<table>
<tr><th>Case #</th><th>Parameter Set</th><th>rx_ber_0</th></tr>
<tr><td>1</td><td>crete2_toolkit_4|tx_vod_0:1, crete2_toolkit_4|tx_test_pattern_0:PRBS7</td><td></td></tr>
<tr><td>2</td><td>crete2_toolkit_4|tx_vod_0:1, crete2_toolkit_4|tx_test_pattern_0:PRBS9</td><td></td></tr>
<tr><td>3</td><td>crete2_toolkit_4|tx_vod_0:2, crete2_toolkit_4|tx_test_pattern_0:PRBS7</td><td></td></tr>
<tr><td>4</td><td>crete2_toolkit_4|tx_vod_0:2, crete2_toolkit_4|tx_test_pattern_0:PRBS9</td><td></td></tr>
<tr><td>5</td><td>crete2_toolkit_4|tx_vod_0:3, crete2_toolkit_4|tx_test_pattern_0:PRBS7</td><td></td></tr>
<tr><td>6</td><td>crete2_toolkit_4|tx_vod_0:3, crete2_toolkit_4|tx_test_pattern_0:PRBS9</td><td></td></tr>
<tr><td>7</td><td>crete2_toolkit_4|tx_vod_0:4, crete2_toolkit_4|tx_test_pattern_0:PRBS7</td><td></td></tr>
<tr><td>8</td><td>crete2_toolkit_4|tx_vod_0:4, crete2_toolkit_4|tx_test_pattern_0:PRBS9</td><td></td></tr>
<tr><td>9</td><td>crete2_toolkit_4|tx_vod_0:5, crete2_toolkit_4|tx_test_pattern_0:PRBS7</td><td></td></tr>
<tr><td>10</td><td>crete2_toolkit_4|tx_vod_0:5, crete2_toolkit_4|tx_test_pattern_0:PRBS9</td><td></td></tr>
<tr><td>11</td><td>crete2_toolkit_4|tx_vod_0:6, crete2_toolkit_4|tx_test_pattern_0:PRBS7</td><td></td></tr>
<tr><td>12</td><td>crete2_toolkit_4|tx_vod_0:6, crete2_toolkit_4|tx_test_pattern_0:PRBS9</td><td></td></tr>
<tr><td>13</td><td>crete2_toolkit_4|tx_vod_0:7, crete2_toolkit_4|tx_test_pattern_0:PRBS7</td><td></td></tr>
<tr><td>14</td><td>crete2_toolkit_4|tx_vod_0:7, crete2_toolkit_4|tx_test_pattern_0:PRBS9</td><td></td></tr>
<tr><td>15</td><td>crete2_toolkit_4|tx_vod_0:8, crete2_toolkit_4|tx_test_pattern_0:PRBS7</td><td></td></tr>
<tr><td>16</td><td>crete2_toolkit_4|tx_vod_0:8, crete2_toolkit_4|tx_test_pattern_0:PRBS9</td><td></td></tr>
<tr><td>17</td><td>crete2_toolkit_4|tx_vod_0:9, crete2_toolkit_4|tx_test_pattern_0:PRBS7</td><td></td></tr>
<tr><td>18</td><td>crete2_toolkit_4|tx_vod_0:9, crete2_toolkit_4|tx_test_pattern_0:PRBS9</td><td></td></tr>
<tr><td>19</td><td>crete2_toolkit_4|tx_vod_0:10, crete2_toolkit_4|tx_test_pattern_0:PRBS7</td><td></td></tr>
<tr><td>20</td><td>crete2_toolkit_4|tx_vod_0:10, crete2_toolkit_4|tx_test_pattern_0:PRBS9</td><td></td></tr>
</table>

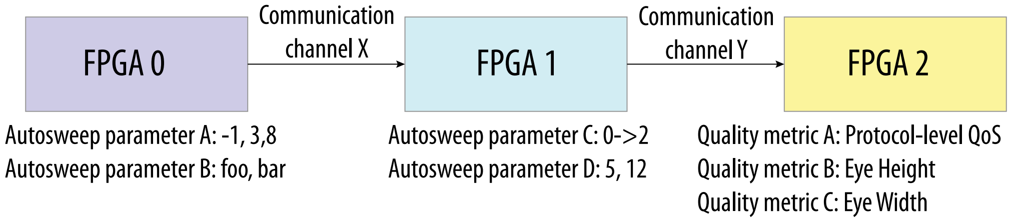
By default, the **Autosweep** view launches without any connection to a toolkit instance(s) or channel pair(s). You can add parameters by clicking **Add Parameter** and selecting parameters from specific toolkit instance(s) in the **Select Parameter** dialog box. You can remove parameters by selecting them and clicking **Remove Parameter**. Alternatively, you can add your own parameters and create as many **Autosweep** views as you want, to allow sweeping over different parameters on different channels of the same instance, or different instances entirely.

To save a parameter set for future use, select the parameter set, and then click **Export Settings**. To load a collection, click **Import Settings**.

Important: Any channel of a particular instance that has parameters currently being swept over in one **Autosweep** view cannot have other (or the same) parameters swept over in a different **Autosweep** view. For example, if one **Autosweep** view is currently sweeping over parameters from `InstA | Channel 0`, and another **Autosweep** view has parameters from `InstA | Channel 0`, an error is displayed if you attempt to start the second sweep before the first has completed. This prevents you from changing more things than are expected from a given run of autosweep.

Consider the following example complex system with parameters spread across multiple devices:

Figure 117. An Example of the Autosweep System


FPGA 0
Communication channel X
FPGA 1
Communication channel Y
FPGA 2
Autosweep parameter A: -1, 3,8
Autosweep parameter B: foo, bar
Autosweep parameter C: 0->2
Autosweep parameter D: 5, 12
Quality metric A: Protocol-level QoS
Quality metric B: Eye Height
Quality metric C: Eye Width

The **Autosweep** view allows you to sweep such a complex system when the multiple devices are visible to System Console. You can select the quality metrics from instances different from those you sweep. You can even span levels of the hardware stack from the PMA-level up to protocol-level signaling.

Results

The **Results** table is populated with one row per autosweep iteration. For every output quality metric added in the **Output Metrics** section, a column for that metric is added to the **Results** table, with new row entries added to the bottom. This format allows sorting of the results by quality metric of the system under test, across many combinations of parameters, to determine which parameter settings achieve best real-world results.

The **Results** table allows visualizing or copying the parameter settings associated with a given case, and sorting by quality metrics. Sort the rows of the table by clicking on the column headers.

Control

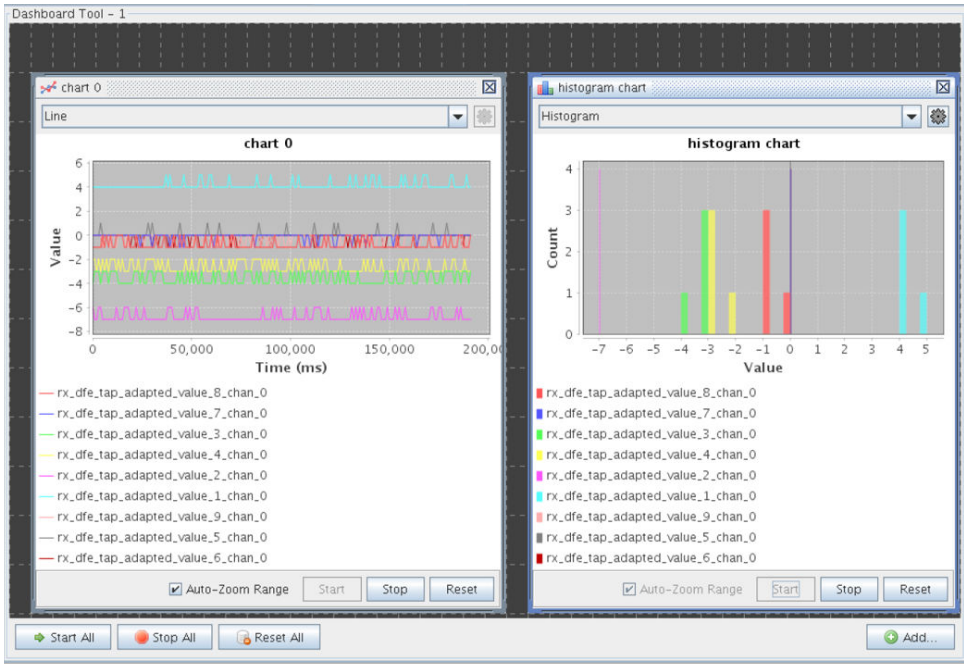
The **Control** pane of the **Autosweep** view allows starting an autosweep run, once you define at least one input parameter and one quality metric. Starting a run, allowing all combinations to complete, and then pressing the **Start** button re-runs the same test case. Pressing the **Stop** button cancels a currently running autosweep.

7.3.1.3. Dashboard View

The **Dashboard** view allows you to visualize the changes to toolkit parameters over time.

The **Dashboard** provides options to view a line chart, histogram, pie chart or bar graph, and a data history. There is no limit imposed on the number of instances of the **Dashboard** view open at once. However, a performance penalty occurs if you update a high number of parameters at a high frequency simultaneously.

Figure 118. Dashboard View of the System Console


Dashboard Tool - 1
chart 0
Line
chart 0
6
4
2
0
-2
-4
-6
-8
Value
0
50,000
100,000
150,000
200,00
Time (ms)
rx_dfe_tap_adapted_value_8_chan_0
rx_dfe_tap_adapted_value_7_chan_0
rx_dfe_tap_adapted_value_3_chan_0
rx_dfe_tap_adapted_value_4_chan_0
rx_dfe_tap_adapted_value_2_chan_0
rx_dfe_tap_adapted_value_1_chan_0
rx_dfe_tap_adapted_value_9_chan_0
rx_dfe_tap_adapted_value_5_chan_0
rx_dfe_tap_adapted_value_6_chan_0
☑ Auto-Zoom Range
Start
Stop
Reset
histogram chart
Histogram
histogram chart
4
3
2
1
0
Count
-7
-6
-5
-4
-3
-2
-1
0
1
2
3
4
5
Value
rx_dfe_tap_adapted_value_8_chan_0
rx_dfe_tap_adapted_value_7_chan_0
rx_dfe_tap_adapted_value_3_chan_0
rx_dfe_tap_adapted_value_4_chan_0
rx_dfe_tap_adapted_value_2_chan_0
rx_dfe_tap_adapted_value_1_chan_0
rx_dfe_tap_adapted_value_9_chan_0
rx_dfe_tap_adapted_value_5_chan_0
rx_dfe_tap_adapted_value_6_chan_0
☑ Auto-Zoom Range
Start
Stop
Reset
Start All
Stop All
Reset All
Add...

The **Add Parameter** dialog box opens when you click the **Add** button. Only those parameters that declare the `allows_charting` parameter property are available for selection in the **Add Parameter** dialog box.

7.3.1.4. Eye Viewer

The System Console **Eye Viewer** allows you to configure, run, and render eye scans. The **Eye Viewer** allows independent control of the eye for each transceiver instance. System Console allows you to open only one **Eye Viewer** per-instance channel pair at any given time. Therefore, there is a one-to-one mapping of a given **Eye Viewer** GUI to a given instance of the eye capture hardware on the FPGA. Click **Tools ➤ Eye Viewer** to launch the **Eye Viewer**.

Eye Viewer Controls

The **Eye Viewer** controls allow you to configure toolkit-specific settings for the current **Eye Viewer** scan. The parameters in the **Eye Viewer** affect the behavior and details of the eye scan run.

Start / Stop Controls

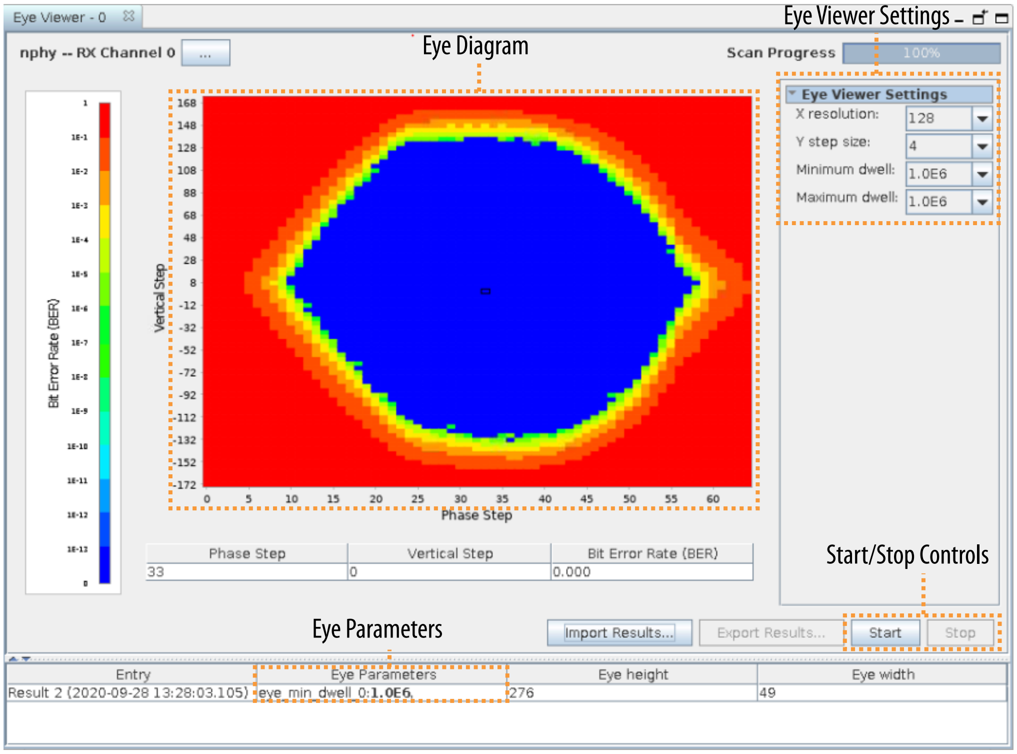
The **Eye Viewer** provides **Start** and **Stop** controls. The **Start** button starts the eye scanning process while the **Stop** button cancels an incomplete scan.

Note: The actual scanning controls, configurations, and metrics shown with the **Eye Viewer** vary by toolkit.

Eye Diagram Visualization

The eye diagram displays the transceiver eye captured from on-die instrumentation (ODI) with a color gradient.

Figure 119. Eye Viewer (E-Tile Transceiver Native PHY Toolkit Example)



nphy -- RX Channel 0 ...

Scan Progress 100%

Eye Viewer Settings
X resolution: 128
Y step size: 4
Minimum dwell: 1.0E6
Maximum dwell: 1.0E6

Phase Step	Vertical Step	Bit Error Rate (BER)
33	0	0.000

Entry	Eye Parameters	Eye height	Eye width
Result 2 (2020-09-28 13:28:03.105)	eye_min_dwell_0:1.0E6	276	49

Results Table

The **Results** table displays results and statistics of all eye scans. While an eye scan is running, you cannot view any partial results. However, there is a progress bar showing the current progress of the eye scan underway.

When an eye scan successfully completes, a new entry appears in the **Results** table, and that entry automatically gains focus. When you select a given entry in the **Results** table, the eye diagram renders the associated eye data. You can right-click in the **Results** table to do the following:

- Apply the test case parameters to the device
- Delete an entry

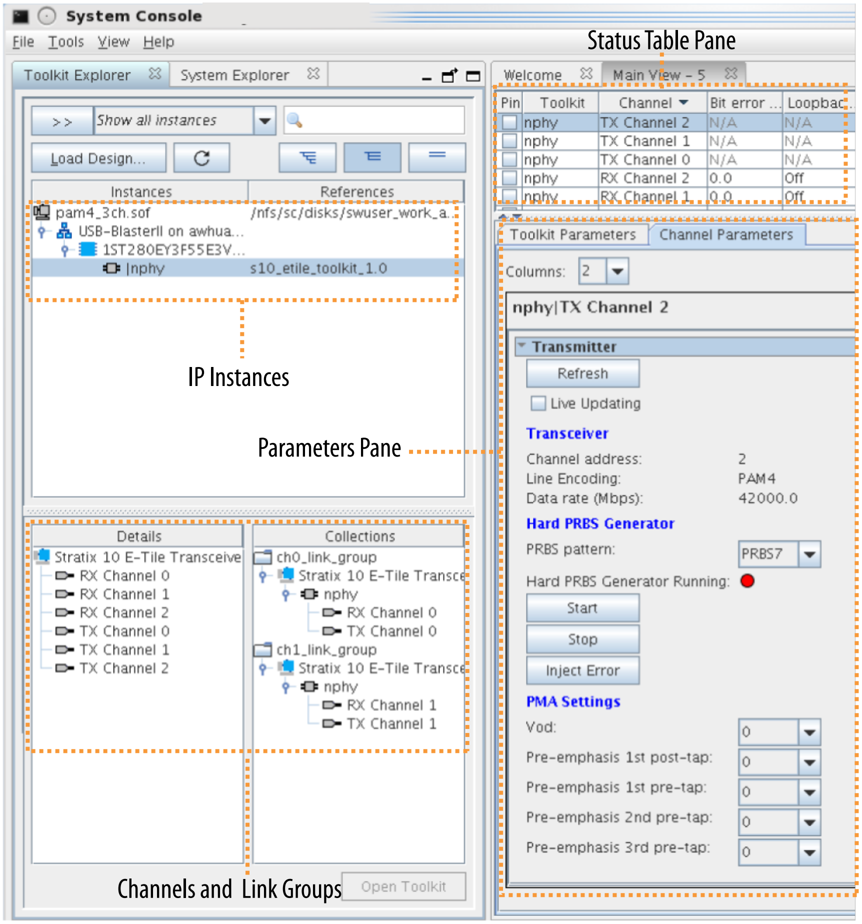
If developing your own toolkit that includes the **Eye Viewer**, the BER gradient is configurable, and the eye diagram GUI supports the following features:

- A BER tool-tip for each cell
- Ability to export the map as PNG
- Zoom

7.3.2. Toolkit Explorer Pane

The **Toolkit Explorer** pane displays all available toolkits and launches tools that use the System Console framework. When you load a design that contains debug-enabled IP, the **Toolkit Explorer** displays the design instances, along with a list of channels and channel collections for debugging. To interact with a channel or a toolkit, double-click on it to launch the **Main View** tabbed window, as shown in the following image:

Figure 120. System Console Toolkit Explorer



Pin	Toolkit	Channel	Bit error ...	Loopbac...
□	nphy	TX Channel 2	N/A	N/A
□	nphy	TX Channel 1	N/A	N/A
□	nphy	TX Channel 0	N/A	N/A
□	nphy	RX Channel 2	0.0	Off
□	nphy	RX Channel 1	0.0	Off

Columns: 2

nphy|TX Channel 2

Transmitter

Refresh

□ Live Updating

Transceiver

Channel address: 2
Line Encoding: PAM4
Data rate (Mbps): 42000.0

Hard PRBS Generator

PRBS pattern: PRBS7

Hard PRBS Generator Running: ●

Start
Stop
Inject Error

PMA Settings

Vod: 0
Pre-emphasis 1st post-tap: 0
Pre-emphasis 1st pre-tap: 0
Pre-emphasis 2nd pre-tap: 0
Pre-emphasis 3rd pre-tap: 0

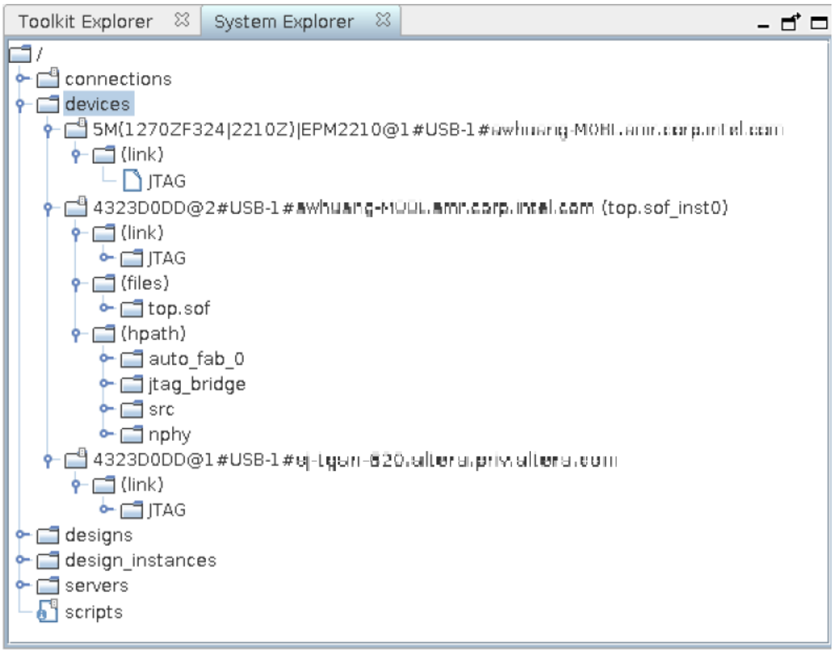
Note: If you close **Toolkit Explorer**, you can reopen it by clicking **View ➤ Toolkit Explorer**.

7.3.3. System Explorer Pane

The **System Explorer** pane displays a list of interactive instances from the design loaded on a connected device. This includes the following items:

- IP instances with debug toolkit capabilities
- IP instances with a debug endpoint

Figure 121. System Explorer Pane



```
/
  connections
  devices
    5M(1270ZF324|2210Z)|EPM2210@1#USB-1#awhuang-MOBL.amr.corp.intel.com
      (link)
        JTAG
    4323D0DD@2#USB-1#awhuang-MOBL.amr.corp.intel.com (top.sof_inst0)
      (link)
        JTAG
      (files)
        top.sof
      (hpath)
        auto_fab_0
        jtag_bridge
        src
        nphy
    4323D0DD@1#USB-1#sj-lgan-820.altera.priv.altera.com
      (link)
        JTAG
  designs
  design_instances
  servers
  scripts
```

Additionally, the **System Explorer** also displays custom toolkit groups and links that you create. **System Explorer** organizes the interactive instances according to the available device connections. The **System Explorer** contains a **Links** instance, and may contain a **Files** instance. The **Links** instance shows debug agents (and other hardware) that System Console can access. The **Files** instance contains information about the programming files loaded from the Intel Quartus Prime project for the device.

The **System Explorer** provides the following information:

- **Devices**—displays information about all devices connected to the System Console.
- **Scripts**—stores scripts for easy execution.
- **Connections**—displays information about the board connections visible to System Console, such as the Intel FPGA Download Cable. Multiple connections are possible.
- **Designs**—displays information about Intel Quartus Prime designs connected to System Console. Each design represents a loaded `.sof` file.
- Right-click on some of the instances to execute related commands.
- Instances that include a message display a message icon. Click on the instance to view the messages in the **Messages** pane.

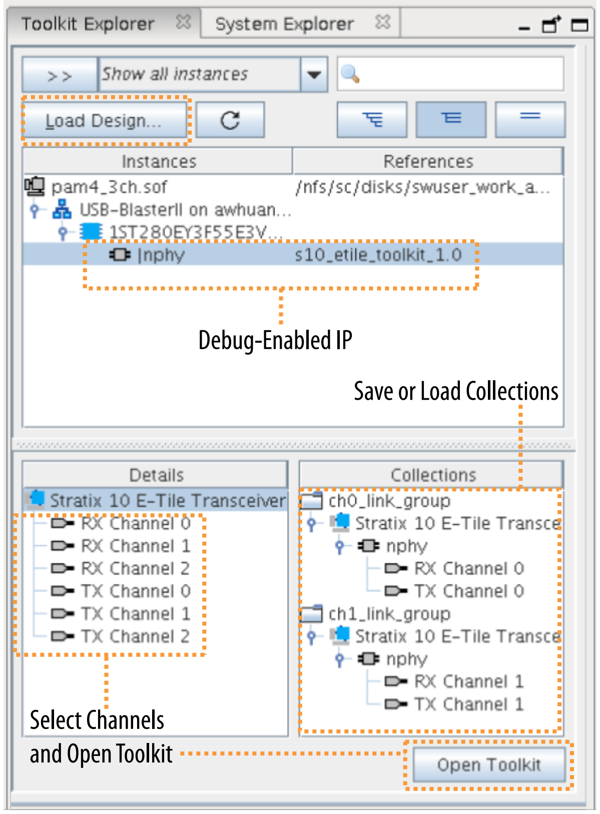
7.4. Launching a Toolkit in System Console

System Console provides the hardware debugging infrastructure to run the Available System Debugging Toolkits on page 174. When you load a design in the **Toolkit Explorer** that includes debug-enabled Intel FPGA IP, the **Toolkit Explorer** automatically lists the toolkits that are available for the IP in the design.

To launch a toolkit in System Console, follow these steps:

1. Create an Intel Quartus Prime project that includes debug-enabled Intel FPGA IP. Refer to IP Cores that Interact with System Console on page 159.
2. On the Compilation Dashboard, double-click **Assembler** to generate a `.sof` programming file for the design. Any prerequisite Compiler stages run automatically before the Assembler starts.
3. Launch System Console, as Starting System Console on page 161 describes.

Figure 122. Launching a Toolkit in System Console



4. In the **Toolkit Explorer**, click **Load Design**, and then select the `.sof` file that you create in step 2. When you load the design, **Toolkit Explorer** displays the debug-enabled IP instances.
5. Select a debug-enabled IP instance. The **Details** pane displays the channels that can launch toolkits.
6. To launch a toolkit, select the toolkit under **Details**. For toolkits with channels, you can also multi-select one or more channels from the **Details** pane. Then, click **Open Toolkit**. The toolkit opens in the **Main View**, and the **Collections** pane displays a collection of any channels that you select.
7. To save a collection for future use, right-click the collection, and then click **Export Collection.** To load a collection, right-click in the **Collections** pane, and then click **Import Collection**. By default, System Console creates a collection when you launch a toolkit.

Related Information

System Console and Toolkit Tcl Command Reference Manual

7.4.1. Available System Debugging Toolkits

The following toolkits are available to launch from the System Console **Toolkit Explorer** in the current version of the Intel Quartus Prime software.

Table 36. Toolkits Available in System Console Toolkit Explorer

Toolkit	Description	Toolkit Documentation
EMIF Calibration Debug Toolkit	Helps you to debug external memory interfaces by accessing calibration data obtained during memory calibration. The analysis tools can evaluate the stability of the calibrated interface and assess hardware conditions.	• *External Memory Interfaces Intel Agilex™ FPGA IP User Guide* • *External Memory Interfaces Intel Stratix 10 FPGA IP User Guide*
EMIF Traffic Generator Configuration Toolkit	Helps you to debug external memory interfaces by sending sample traffic through the external memory interface to the memory device. The generated EMIF design example includes a traffic generator block with control and status registers.	
EMIF Efficiency Monitor Toolkit	Helps you to debug external memory interfaces by measuring efficiency on the Avalon interface in real time. The generated EMIF design example can include the Efficiency Monitor block.	• *External Memory Interfaces Intel Agilex FPGA IP User Guide* • *External Memory Interfaces Intel Stratix 10 FPGA IP User Guide*
Ethernet Toolkit	Helps you to interact with and debug an Ethernet Intel FPGA IP interface in real time. You can verify the status of the Ethernet link, assert and deassert IP resets, verifies the IP error correction capability,	*Ethernet Toolkit User Guide*
Intel Stratix 10 FPGA P-Tile Toolkit (for PCIe)	Helps you to optimize the performance of large-size data transfers with real-time control, monitoring, and debugging of the PCI Express* links at the Physical, Data Link, and Transaction layers.	*Intel FPGA P-Tile Avalon Memory Mapped IP for PCI Express* User Guide*
Serial Lite IV IP Toolkit	An inspection tool that monitors the status of a Serial Lite IV IP link and provides a step-by-step guide for the IP link initialization sequences.	• *Serial Lite IV Intel Agilex FPGA IP Design Example User Guide* • *Serial Lite IV Intel Stratix 10 FPGA IP Design Example User Guide*
Intel Arria 10 and Intel Cyclone 10 GX Transceiver Native PHY Toolkit	Helps you to optimize high-speed serial links in your board design by providing real-time control, monitoring, and debugging of the transceiver links running on your board.	
L-Tile and H-Tile Transceiver Native PHY Toolkit		
E-Tile Transceiver Native PHY Toolkit		

The following legacy toolkits remain available by clicking **Tools ➤ Legacy Toolkits** in System Console. The legacy toolkits support earlier device families and may be subject to end of life and removal of support in a coming software release.

Table 37. Legacy Toolkits Available in System Console

Legacy Toolkit	Description	Legacy Toolkit Documentation
Ethernet Link Inspector - Link Monitor Toolkit	The Ethernet Link Inspector is an inspection tool that can continuously monitor an Ethernet link that contains an Ethernet IP. The Link Monitor toolkit performs real-time status monitoring of an Ethernet IP link. The link monitor continuously reads and displays all of the required status registers related to the Ethernet IP link, and displays the Ethernet IP link status at various stages are valid.	*Ethernet Link Inspector User Guide for Intel Stratix 10 Devices*
Ethernet Link Inspector - Link Analysis Toolkit	The Link Analysis toolkit displays a sequence of events on an Ethernet IP link, which occur in a finite duration of time. The Link Analysis requires the Signal Tap logic analyzer to first capture and store a database (`.csv`) of all required signals. The Link Analysis toolkit then performs an analysis on the database to extract all the required information and displays them in a user-friendly graphical user interface (GUI).	
S10 SDM Debug Toolkit	Provides access to current status of the Intel Stratix 10 device. To use these commands, you must have a valid design loaded that includes the module that you intend to access.	*Intel Stratix 10 Configuration User Guide*

Note: Refer to the toolkit documentation for individual toolkit launch, setup, and use information. The Transceiver Toolkit previously available in the Intel Quartus Prime software Tools menu is replaced by the Intel Arria 10 and Intel Cyclone 10 GX Transceiver Native PHY Toolkit.

Related Information

- External Memory Interfaces Intel Agilex FPGA IP User Guide
- External Memory Interfaces Intel Stratix 10 FPGA IP User Guide
- Ethernet Toolkit User Guide
- Intel FPGA P-Tile Avalon Memory Mapped IP for PCI Express* User Guide
- Intel FPGA P-Tile Avalon Streaming IP for PCI Express* User Guide
- Ethernet Link Inspector User Guide for Intel Stratix 10 Devices
- Intel Stratix 10 Configuration User Guide
- Serial Lite IV Intel Agilex FPGA IP Design Example User Guide
- Serial Lite IV Intel Stratix 10 FPGA IP Design Example User Guide

7.4.2. Creating Collections from the Toolkit Explorer

You can create custom collections to view and configure members from different instances in a single Main View.

Perform these steps to group instances:

1. Select multiple items in the instances tree.
2. Right click to view the context-sensitive menu.
3. Select **Add to Collection ➤ New Collection**. The **Add to Collection** dialog box appears with members you select.

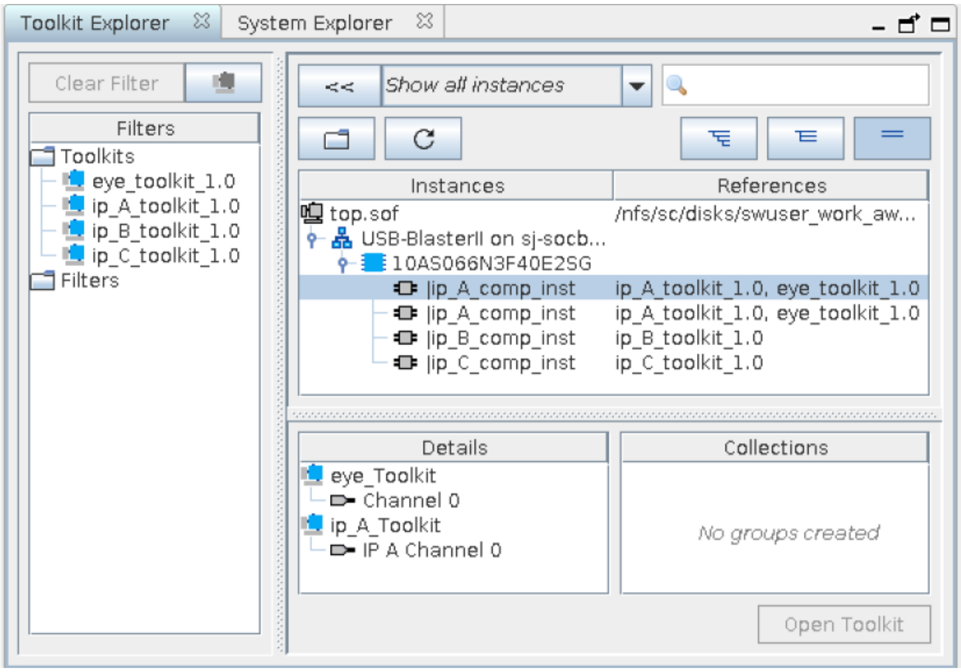
System Console adds the collections that you create to the **Collections** pane of the **Toolkit Explorer**. You can perform one of the following actions:

- Double-click on a custom-created collection to launch the Main view containing all of the group’s members.
- Right-click on an existing collection member and select **Remove from Collection** to remove the member.

7.4.3. Filtering and Searching Interactive Instances

By default, the **Toolkits** list shows all toolkit instances and their respective channels linking to the System Console. This view is useful in simple cases, but can become very dense in a complex system having many debug-enabled IPs, and having potentially multiple FPGAs connected to System Console.

Figure 123. Toolkit Explorer with Filters



Instances	References
top.sof	/nfs/sc/disks/swuser_work_aw...
USB-BlasterII on sj-socb...	
10AS066N3F40E2SG	
|ip_A_comp_inst	ip_A_toolkit_1.0, eye_toolkit_1.0
|ip_A_comp_inst	ip_A_toolkit_1.0, eye_toolkit_1.0
|ip_B_comp_inst	ip_B_toolkit_1.0
|ip_C_comp_inst	ip_C_toolkit_1.0

Details
eye_Toolkit
Channel 0
ip_A_Toolkit
IP A Channel 0

Collections
No groups created

To limit the information display, the **Filter** list allows filtering the display by toolkit types currently available in the System Console. You can also create custom filters using groups, for example, “Inst A, Inst F, and Inst Z”, or “E-Tile and L/H-Tile Transceivers only".

To refine the list of toolkits, use the search field in the **Toolkit Explorer** to filter the list further.

7.5. Using System Console Services

System Console services provide access to hardware modules that you instantiate in your FPGA. Services vary in the type of debug access they provide.

Related Information

System Console and Toolkit Tcl Command Reference Manual

7.5.1. Locating Available Services

System Console uses a virtual file system to organize the available services, which is similar to the `/dev location` on Linux systems. Board connection, device type, and IP names are all part of a service path. Instances of services are referred to by their unique service path in the file system. To retrieve service paths for a particular service, use the command `get_service_paths` <*service-type*>.

Example 11. Locating a Service Path

```
#We are interested in master services.
set service_type "master"

#Get all the paths as a list.
set master_service_paths [get_service_paths $service_type]

#We are interested in the first service in the list.
set master_index 0

#The path of the first master.
set master_path [lindex $master_service_paths $master_index]

#Or condense the above statements into one statement:
set master_path [lindex [get_service_paths master] 0]
```

System Console commands require service paths to identify the service instance you want to access. The paths for different components can change between runs of System Console and between versions. Use the `get_service_paths` command to obtain service paths.

The string values of service paths change with different releases of the tool. Use the `marker_node_info` command to get information from the path.

System Console automatically discovers most services at startup. System Console automatically scans for all JTAG and USB-based service instances and retrieves their service paths. System Console does not automatically discover some services, such as TCP/IP. Use `add_service` to inform System Console about those services.

Example 12. Marker_node_info

Use the `marker_node_info` command to get information about SLD nodes associated with the specified service.

```
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)
```

7.5.2. Opening and Closing Services

After you have a service path to a particular service instance, you can access the service for use.

The `claim_service` command directs System Console to start using a particular service instance, and with no additional arguments, claims a service instance for exclusive use.

Example 13. 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 14. Closing a Service

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

7.5.3. Using the 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 15. 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 at 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

Virtual JTAG IP Core User Guide

7.5.3.1. SLD Commands

Table 38. 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 returning 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

System Console and Toolkit Tcl Command Reference Manual

7.5.4. Using the In-System Sources and Probes Service

The In-System Sources and Probes (ISSP) service provides scriptable access to the In-System Sources and Probes Intel FPGA IP in a similar manner to using the **In-System Sources and Probes Editor** in the Intel Quartus Prime software.

Example 16. ISSP Service

Before you use the ISSP service, ensure your design works in the **In-System Sources and Probes Editor**. In System Console, open the service for an ISSP instance:

```
set issp_index 0
set issp [lindex [get_service_paths issp] 0]
set claimed_issp [claim_service issp $issp mylib]
```

View information about this particular ISSP instance:

```
array set instance_info [issp_get_instance_info $claimed_issp]
set source_width $instance_info(source_width)
set probe_width $instance_info(probe_width)
```

The Intel Quartus Prime software reads probe data as a single bitstring of length equal to the probe width:

```
set all_probe_data [issp_read_probe_data $claimed_issp]
```

As an example, you can define the following procedure to extract an individual probe line's data:

```
proc get_probe_line_data {all_probe_data index} {
    set line_data [expr { ($all_probe_data >> $index) & 1 }]
    return $line_data
}
set initial_all_probe_data [issp_read_probe_data $claim_issp]
set initial_line_0 [get_probe_line_data $initial_all_probe_data 0]
set initial_line_5 [get_probe_line_data $initial_all_probe_data 5]
# ...
set final_all_probe_data [issp_read_probe_data $claimed_issp]
set final_line_0 [get_probe_line_data $final_all_probe_data 0]
```

Similarly, the Intel Quartus Prime software writes source data as a single bitstring of length equal to the source width:

```
set source_data 0xDEADBEEF
issp_write_source_data $claimed_issp $source_data
```

You can also retrieve the currently set source data:

```
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
```

7.5.4.1. In-System Sources and Probes Commands

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

Table 39. 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

System Console and Toolkit Tcl Command Reference Manual

7.5.5. Using the Monitor Service

The monitor service builds on top of the host service to allow reads of Avalon memory-mapped interface agents 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 host service manually for the reads.

Example 17. Monitor Service

1. Determine the host and the memory address range that you want to poll:

```
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
```

   With the first host, read 100 bytes starting at address 0x2000 every 100 milliseconds.

2. Open the monitor service:

```
set monitor [lindex [get_service_paths monitor] 0]
set claimed_monitor [claim_service monitor $monitor mylib]
```

   The monitor service opens the host service automatically.

3. With the monitor service, register the address range and time interval:

```
monitor_add_range $claimed_monitor $master $address $bytes_to_read
monitor_set_interval $claimed_monitor $read_interval_ms
```

4. Add more ranges, defining the result at each interval:

```
global monitor_data_buffer
set monitor_data_buffer [list]
```

5. Gather the data and append it with a global variable:

```
proc store_data {monitor master address bytes_to_read} {\
  global monitor_data_buffer
# monitor_read_data returns the range of data polled from the running \
  design as a list
#(in this example, a 100-element list).
  set data [monitor_read_data $claimed_monitor $master $address \
  $bytes_to_read]
# Append the list as a single element in the monitor_data_buffer \
  global list.
  lappend monitor_data_buffer $data
}
```

   Note: 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` returns the latest polled data.

6. 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
```

7. 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.

7.5.5.1. Monitor Commands

You can use the Monitor commands to read many Avalon memory-mapped interface agent 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 40. 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 host 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 host service. *<address>* and *<size>* are relative to the host service.
`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_interval`	*<service-path>*	Returns the current interval set which specifies the frequency of the polling action.
`monitor_get_missing_event_count`	*<service-path>*	Returns the number of callback events missed during the evaluation of last Tcl callback expression.
`monitor_get_read_interval`	*<service-path>* *<target-path>* *<address>* *<size>*	Returns the milliseconds elapsed between last two data reads returned by `monitor_read_data`.

continued...

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. You must specify a memory range within the range in `monitor_add_range`.
`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. You must specify a memory range within the range in `monitor_add_range`.
`monitor_set_callback`	*<service-path>* *<Tcl-expression>*	Specifies a Tcl expression that the System Console must evaluate after reading all the memories that this service monitors. Typically, you specify this expression as a single string Tcl procedure call with necessary argument passed in.
`monitor_set_enabled`	*<service-path>* *<enable(1)/disable(0)>*	Enables and disables monitoring. Memory read starts after this command, and Tcl callback evaluates after data is read.
`monitor_set_interval`	*<service-path>* *<interval>*	Defines the target frequency of the polling action by specifying the interval between two memory reads. The actual polling frequency varies depending on the system activity.

Related Information

System Console and Toolkit Tcl Command Reference Manual

7.5.6. Using the Device Service

The device service supports device-level actions.

Example 18. 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. host 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
```

7.5.6.1. Device Commands

The device commands provide access to programmable logic devices on your board. Before you use these commands, identify the path to the programmable logic device on your board using the `get_service_paths`.

Table 41. 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

System Console and Toolkit Tcl Command Reference Manual

7.5.7. Using the Design Service

You can use design service commands to work with Intel Quartus Prime design information.

Example 19. Load

When you open System Console from the Intel Quartus Prime software, the current project's debug information is sourced automatically if the `.sof` file is present. In other situations, you can load the `.sof` manually.

```
set sof_path [file join project_dir output_files project_name.sof]
set design [design_load $sof_path]
```

System Console is now aware of the `.sof` loading.

Example 20. Linking

Once a `.sof` loads, System Console automatically links design information to the connected device. The link persists and you can unlink or reuse the link on an equivalent device with the same `.sof`.

You can perform manual linking as follows:

```
set device_index 0; # Device index for our target
set device [lindex [get_service_paths device] $device_index]
design_link $design $device
```

Manual 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`.

7.5.7.1. Design Service Commands

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

Table 42. Design Service Commands

Command	Arguments	Function
`design_load`	*<quartus-project-path>*, *<sof-file-path>*,	Loads a model of an Intel Quartus Prime design into System Console. Returns the design path.

continued...

Command	Arguments	Function
	or *<qpf-file-path>*	For example, if your Intel Quartus Prime Project File (`.qpf`) is in `c:/projects/loopback`, type the following command: `design_load {c:\projects\loopback\}`
`design_link`	*<design-path>* *<device-service-path>*	Links an Intel Quartus Prime logical design with a physical device. For example, you can link an Intel Quartus Prime design called **2c35_quartus_design** to a 2c35 device. After you create this link, System Console creates the appropriate correspondences between the logical and physical submodules of the Intel Quartus Prime project.
`design_extract_debug_files`	*<design-path>* *<zip-file-name>*	Extracts debug files from a `.sof` to a zip file which can be emailed to *Intel FPGA Support* for analysis. You can specify a design path of `{}` to unlink a device and to disable auto linking for that device.
`design_get_warnings`	*<design-path>*	Gets the list of warnings for this design. If the design loads correctly, then an empty list returns.

Related Information

System Console and Toolkit Tcl Command Reference Manual

7.5.8. Using the Bytestream Service

The bytestream service provides access to modules that produce or consume a stream of bytes. Use the bytestream service to communicate directly to the IP core that provides bytestream interfaces, such as the JTAG UART or the Avalon Streaming JTAG interface Intel FPGA IP.

Example 21. 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
```

7.5.8.1. Bytestream Commands

Table 43. 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

System Console and Toolkit Tcl Command Reference Manual

7.5.9. Using the 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]
```

7.5.9.1. JTAG Debug Commands

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

Table 44. 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. This command 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 clock signal of the system clock that drives the module's system interface. The clock value is sampled asynchronously; consequently, you must sample the clock several times to guarantee that it is switching.

continued...

Command	Argument	Function
jtag_debug_sample_reset	*<service-path>*	Returns the value of the reset_n signal of the Avalon-ST JTAG Interface core. If reset_n is low (asserted), the value is 0 and if reset_n is high (deasserted), the value is 1.
jtag_debug_sense_clock	*<service-path>*	Returns a sticky bit that monitors system clock activity. If the clock switched at least once since the last execution of this command, returns 1. Otherwise, returns 0.. 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

System Console and Toolkit Tcl Command Reference Manual

7.6. On-Board Intel FPGA Download Cable II Support

System Console supports an On-Board Intel FPGA Download Cable II circuit via the USB Debug Master IP component. This IP core supports the master service.

7.7. MATLAB and Simulink* in a System Verification Flow

You can test system development in System Console using MATLAB and Simulink*, and set up a system verification flow using the Intel FPGA Hardware in the Loop (HIL) tools. In this approach, you deploy the design hardware to run in real time, and simulate the system's surrounding components in a software environment. The HIL approach allows you to use the flexibility of software tools with the real-world accuracy and speed of hardware. You can gradually introduce more hardware components to the system verification testbench. This technique gives you more control over the integration process as you tune and validate the system. When the full system is integrated, the HIL approach allows you to provide stimuli via software to test the system under a variety of scenarios.

Advantages of HIL Approach

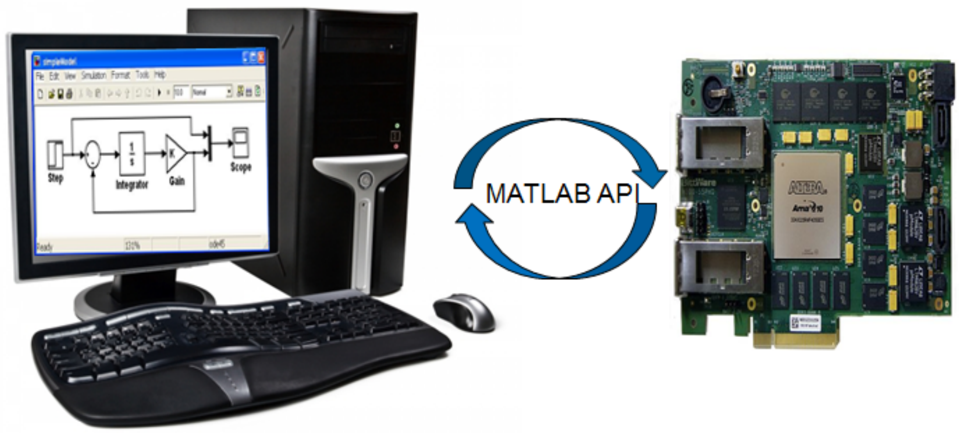
- Avoid long computational delays for algorithms with high processing rates
- API helps to control, debug, visualize, and verify FPGA designs all within the MATLAB environment
- FPGA results are read back by the MATLAB software for further analysis and display

Required Tools and Components

- MATLAB software
- DSP Builder for Intel FPGAs software
- Intel Quartus Prime software
- Intel FPGA

Note: The DSP Builder for Intel FPGAs installation bundle includes the System Console MATLAB API.

Figure 124. Hardware in the Loop Host-Target Setup



Related Information

Hardware in the Loop from the MATLAB Simulink Environment white paper

7.7.1. Supported MATLAB API Commands

You can perform the work from the MATLAB environment, and read and write to hosts and agents through the System Console. The supported MATLAB API commands do not require launching the System Console GUI. The supported commands are:

- `SystemConsole.refreshMasters;`
- `M = SystemConsole.openMaster(1);`
- `M.write (type, byte address, data);`
- `M.read (type, byte address, number of words);`
- `M.close`

Example 22. 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
```

7.7.2. High Level Flow

1. Install the DSP Builder for Intel FPGAs software, so you have the necessary libraries to enable this flow
2. Build the design using Simulink and the DSP Builder for Intel FPGAs libraries.

   DSP Builder for Intel FPGAs helps to convert the Simulink design to HDL
3. Include Avalon memory mapped components in the design (DSP Builder for Intel FPGAs can port non-Avalon memory mapped components)

4. Include Signals and Control blocks in the design
5. Separate synthesizable and non-synthesizable logic with boundary blocks.
6. Integrate the DSP system in Platform Designer
7. Program the Intel FPGA
8. Interact with the Intel FPGA through the supported MATLAB API commands.

7.8. System Console Examples and Tutorials

Intel provides examples for performing board bring-up, creating a simple toolkit, and programming a Nios II processor. The `System_Console.zip` file contains design files for the board bring-up example. The Nios II Ethernet Standard `.zip` files contain the design files for the Nios II processor example.

Note: The instructions for these examples assume that you are familiar with the Intel Quartus Prime software, Tcl commands, and Platform Designer.

Related Information

On-Chip Debugging Design Examples Website
Contains the design files for the example designs that you can download.

7.8.1. 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 Intel 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

- Nios II Ethernet Standard Design Example
- Nios II Gen2 Software Developer's Handbook

7.8.1.1. Processor Commands

Table 45. Processor Commands

Command [2]	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>*	Returns the value of the specified register.

continued...

[2] 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.

Command [2]	Arguments	Function
	<register_name>	
`processor_set_register`	*<service-path>* *<register_name>* *<value>*	Sets the value of the specified register.

Related Information

Nios II Processor Example on page 189

7.9. Running System Console in Command-Line Mode

You can run System Console in command line mode and either work interactively or run a Tcl script. System Console prints the output in the console window.

- `--cli`—Runs System Console in command-line mode.
- `--project_dir=<`*`project dir`*`>`—Directs System Console to the location of your hardware project. Also works in GUI mode.
- `--script=<`*`your script`*`>.tcl`—Directs System Console to run your Tcl script.
- `--help`— Lists all available commands. Typing `--help` *<command name>* provides the syntax and arguments of the command.

System Console provides command completion if you type the beginning letters of a command and then press the **Tab** key.

Related Information

System Console and Toolkit Tcl Command Reference Manual

7.10. Using System Console Commands

You can use System Console commands to control hardware debug and testing with the command-line or scripting. Use System Console commands to identify a System Console service by its path, to open and close a connection, add a service, and a variety of other System Console controls.

Note: For a complete reference of currently supported System Console and toolkit commands, refer to the *System Console and Toolkit Tcl Command Reference Manual*.

The following steps show initiation of a simple service connection:

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

[2] 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.

Related Information

System Console and Toolkit Tcl Command Reference Manual

7.11. Using Toolkit Tcl Commands

For advanced users, System Console also supports Tcl commands that allow you to define and operate your own custom toolkits.

You can use the Toolkit Tcl commands to add and set the toolkit requirements and properties, and to retrieve accessible toolkit modules, systems, and services at the command-line or with scripting.

Note: For a complete reference of currently supported System Console and toolkit commands, refer to the *System Console and Toolkit Tcl Command Reference Manual*.

Related Information

System Console and Toolkit Tcl Command Reference Manual

7.12. Analyzing and Debugging Designs with the System Console Revision History

The following revision history applies to this chapter:

Document Version	Intel Quartus Prime Version	Changes
2021.06.21	21.2	• Moved System Console and Toolkit Tcl command descriptions to *System Console and Toolkit Tcl Command Reference Manual* and provided links to this new comprehensive document. • Replaced non-inclusive terms with "host" and "agent" inclusive terms for Avalon memory mapped interface descriptions and related GUI elements. • Added toolkit definition to *Introduction to System Console* topic. • Revised *System Console Tools* figure. • Revised wording of *Autosweep View* topic for clarity. • Added details to explanation of legacy toolkits in *Available System Debugging Toolkits* • Added ISSP service to *Common Services for System Console* table. • Added link to download center.
2021.03.29	21.1	• Added link to *Introduction to System Console* topic.
2020.09.28	20.3	• Revised "Introduction to System Console" wording and block diagram. • Revised "Starting System Console" to consolidate all methods. • Revised "Toolkit Explorer Pane" to refer to launching toolkits. • Revised "Autosweep View" to account for use with or without toolkit and export and import of settings. • Added new "Launching a Toolkit in System Console" topic. • Added new "Available System Debugging Toolkits" topic. • Added new Toolkit Tcl Commands section. • Reordered some topics and updated outdated screenshots.
2019.09.30	19.3	Made the following updates in the *Analyzing and Debugging Designs with System Console* chapter:

continued...

Document Version	Intel Quartus Prime Version	Changes
		• Updated *System Console GUI* and *System Explorer Pane* topics to describe the new framework. • Added the following new topics to describe various panes and views added to the System Console: — *System Console Default Panes* — *Toolkit Explorer Pane* — *Filtering and Searching Interactive Instances* — *Creating Collections from the Toolkit Explorer* — *System Console Views* — *Main View* — *Link Pair View* — *Autosweep View* — *Dashboard View* — *Eye View* • Removed *Working with Toolkit* section completely since it was now outdated due to the implementation of new System Console framework.
2018.05.07	18.0.0	Removed obsolete section: *Board Bring-Up with System Console Tutorial*.
2017.05.08	17.0.0	• Created topic *Convert your Dashboard Scripts to Toolkit API*. • Removed *Registering the Service* Example from *Toolkit API Script Examples*, and added corresponding code snippet to *Registering a Toolkit*. • Moved *.toolkit Description File Example* under *Creating a Toolkit Description File*. • Renamed *Toolkit API GUI Example .toolkit File* to *.toolkit Description File Example*. • Updated examples on Toolkit API to reflect current supported syntax.
2016.10.31	16.1.0	• Implemented Intel rebranding.
2015.11.02	15.1.0	• Edits to Toolkit API content and command format. • Added Toolkit API design example. • Added graphic to *Introduction to System Console*. • Deprecated Dashboard. • Changed instances of *Quartus II* to *Intel Quartus Prime*.
October 2015	15.1.0	• Added content for Toolkit API — Required .toolkit and Tcl files — Registering and launching the toolkit — Toolkit discovery and matching toolkits to IP — Toolkit API commands table
May 2015	15.0.0	Added information about how to download and start System Console stand-alone.
December 2014	14.1.0	• Added overview and procedures for using ADC Toolkit on MAX 10 devices. • Added overview for using MATLAB/Simulink Environment with System Console for system verification.
June 2014	14.0.0	Updated design examples for the following: board bring-up, dashboard service, Nios II processor, design service, device service, monitor service, bytestream service, SLD service, and ISSP service.
November 2013	13.1.0	Re-organization of sections. Added high-level information with block diagram, workflow, SLD overview, use cases, and example Tcl scripts.
June 2013	13.0.0	Updated Tcl command tables. Added board bring-up design example. Removed SOPC Builder content.

continued...

Document Version	Intel Quartus Prime Version	Changes
November 2012	12.1.0	Re-organization of content.
August 2012	12.0.1	Moved Transceiver Toolkit commands to Transceiver Toolkit chapter.
June 2012	12.0.0	Maintenance release. This chapter adds new System Console features.
November 2011	11.1.0	Maintenance release. This chapter adds new System Console features.
May 2011	11.0.0	Maintenance release. This chapter adds new System Console features.
December 2010	10.1.0	Maintenance release. This chapter adds new commands and references for Qsys.
July 2010	10.0.0	Initial release. Previously released as the System Console User Guide, which is being obsoleted. This new chapter adds new commands.

8. Intel Quartus Prime Pro Edition User Guide Debug Tools Archives

If the table does not list a software version, the user guide for the previous software version applies.

Intel Quartus Prime Version	User Guide
21.2	Intel Quartus Prime Pro Edition User Guide Debug Tools
21.1	Intel Quartus Prime Pro Edition User Guide Debug Tools
20.3	Intel Quartus Prime Pro Edition User Guide Debug Tools
19.3	Intel Quartus Prime Pro Edition User Guide Debug Tools
18.1	Intel Quartus Prime Pro Edition User Guide Debug Tools
18.0	Intel Quartus Prime Pro Edition User Guide Debug Tools

A. Intel Quartus Prime Pro Edition User Guides

Refer to the following user guides for comprehensive information on all phases of the Intel Quartus Prime Pro Edition FPGA design flow.

Related Information

- Intel Quartus Prime Pro Edition User Guide: Getting Started
  Introduces the basic features, files, and design flow of the Intel Quartus Prime Pro Edition software, including managing Intel Quartus Prime Pro Edition projects and IP, initial design planning considerations, and project migration from previous software versions.
- Intel Quartus Prime Pro Edition User Guide: Platform Designer
  Describes creating and optimizing systems using Platform Designer, a system integration tool that simplifies integrating customized IP cores in your project. Platform Designer automatically generates interconnect logic to connect intellectual property (IP) functions and subsystems.
- Intel Quartus Prime Pro Edition User Guide: Design Recommendations
  Describes best design practices for designing FPGAs with the Intel Quartus Prime Pro Edition software. HDL coding styles and synchronous design practices can significantly impact design performance. Following recommended HDL coding styles ensures that Intel Quartus Prime Pro Edition synthesis optimally implements your design in hardware.
- Intel Quartus Prime Pro Edition User Guide: Design Compilation
  Describes set up, running, and optimization for all stages of the Intel Quartus Prime Pro Edition Compiler. The Compiler synthesizes, places, and routes your design before generating a device programming file.
- Intel Quartus Prime Pro Edition User Guide: Design Optimization
  Describes Intel Quartus Prime Pro Edition settings, tools, and techniques that you can use to achieve the highest design performance in Intel FPGAs. Techniques include optimizing the design netlist, addressing critical chains that limit retiming and timing closure, optimizing device resource usage, device floorplanning, and implementing engineering change orders (ECOs).
- Intel Quartus Prime Pro Edition User Guide: Programmer
  Describes operation of the Intel Quartus Prime Pro Edition Programmer, which allows you to configure Intel FPGA devices, and program CPLD and configuration devices, via connection with an Intel FPGA download cable.
- Intel Quartus Prime Pro Edition User Guide: Block-Based Design
  Describes block-based design flows, also known as modular or hierarchical design flows. These advanced flows enable preservation of design blocks (or logic that comprises a hierarchical design instance) within a project, and reuse of design blocks in other projects.

- Intel Quartus Prime Pro Edition User Guide: Partial Reconfiguration
  Describes Partial Reconfiguration, an advanced design flow that allows you to reconfigure a portion of the FPGA dynamically, while the remaining FPGA design continues to function. Define multiple personas for a particular design region, without impacting operation in other areas.
- Intel Quartus Prime Pro Edition User Guide: Third-party Simulation
  Describes RTL- and gate-level design simulation support for third-party simulation tools by Aldec*, Cadence*, Siemens EDA, and Synopsys* that allow you to verify design behavior before device programming. Includes simulator support, simulation flows, and simulating Intel FPGA IP.
- Intel Quartus Prime Pro Edition User Guide: Third-party Synthesis
  Describes support for optional synthesis of your design in third-party synthesis tools by Siemens EDA, and Synopsys*. Includes design flow steps, generated file descriptions, and synthesis guidelines.
- Intel Quartus Prime Pro Edition User Guide: Third-party Logic Equivalence Checking Tools
  Describes support for optional logic equivalence checking (LEC) of your design in third-party LEC tools by OneSpin*.
- Intel Quartus Prime Pro Edition User Guide: Debug Tools
  Describes a portfolio of Intel Quartus Prime Pro Edition in-system design debugging tools for real-time verification of your design. These tools provide visibility by routing (or "tapping") signals in your design to debugging logic. These tools include System Console, Signal Tap logic analyzer, system debugging toolkits, In-System Memory Content Editor, and In-System Sources and Probes Editor.
- Intel Quartus Prime Pro Edition User Guide: Timing Analyzer
  Explains basic static timing analysis principals and use of the Intel Quartus Prime Pro Edition Timing Analyzer, 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.
- Intel Quartus Prime Pro Edition User Guide: Power Analysis and Optimization
  Describes the Intel Quartus Prime Pro Edition Power Analysis tools that allow accurate estimation of device power consumption. Estimate the power consumption of a device to develop power budgets and design power supplies, voltage regulators, heat sink, and cooling systems.
- Intel Quartus Prime Pro Edition User Guide: Design Constraints
  Describes timing and logic constraints that influence how the Compiler implements your design, such as pin assignments, device options, logic options, and timing constraints. Use the Interface Planner to prototype interface implementations, plan clocks, and quickly define a legal device floorplan. Use the Pin Planner to visualize, modify, and validate all I/O assignments in a graphical representation of the target device.
- Intel Quartus Prime Pro Edition User Guide: PCB Design Tools
  Describes support for optional third-party PCB design tools by Siemens EDA and Cadence*. Also includes information about signal integrity analysis and simulations with HSPICE and IBIS Models.
- Intel Quartus Prime Pro Edition User Guide: Scripting
  Describes use of Tcl and command line scripts to control the Intel Quartus Prime Pro Edition software and to perform a wide range of functions, such as managing projects, specifying constraints, running compilation or timing analysis, or generating reports.