14 Design Debugging with the Signal Tap Logic Analyzer

14.1 About the Signal Tap Logic Analyzer

The Signal Tap Logic Analyzer is a next-generation, system-level debugging tool that captures and displays real-time signal behavior in an FPGA design. You can examine the behavior of internal signals without using extra I/O pins, while the design is running at full speed on an FPGA.

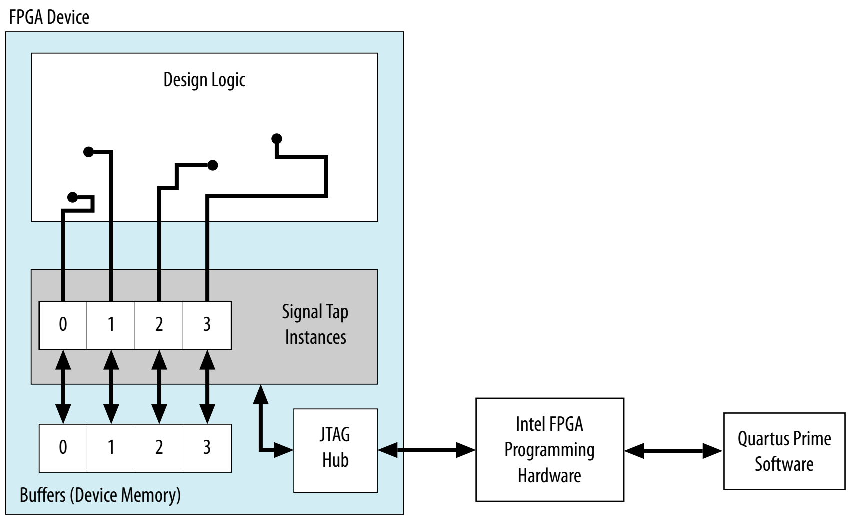
The Signal Tap Logic Analyzer is scalable, easy to use, and available as a stand-alone package or with a software subscription.

The Signal Tap Logic Analyzer supports these features:

- Debug an FPGA design by probing the state of internal signals without the need of external equipment.
- Define custom trigger-condition logic for greater accuracy and improved ability to isolate problems.
- Capture the state of internal nodes or I/O pins in the design without the need of design file changes.
- Store all captured signal data in device memory until you are ready to read and analyze it.

The Signal Tap Logic Analyzer supports the highest number of channels, largest sample depth, and fastest clock speeds of any logic analyzer in the programmable logic market.

Figure 161. Signal Tap Logic Analyzer Block Diagram



Note to figure:

1. This diagram assumes that you compiled the Signal Tap Logic Analyzer with the design as a separate design partition using the Intel Quartus Prime incremental compilation feature. If you do not use incremental compilation, the Compiler integrates the Signal Tap logic with the design.

This chapter is intended for any designer who wants to debug an FPGA design during normal device operation without the need for external lab equipment. Because the Signal Tap Logic Analyzer is similar to traditional external logic analyzers, familiarity with external logic analyzer operations is helpful, but not necessary. To take advantage of faster compile times when making changes to the Signal Tap Logic Analyzer, knowledge of the Intel Quartus Prime incremental compilation feature is helpful.

14.1.1 Hardware and Software Requirements

You need the following hardware and software to perform logic analysis with the Signal Tap Logic Analyzer:

- Signal Tap Logic Analyzer software
- Download/upload cable
- Intel development kit or your design board with JTAG connection to device under test

You can use the Signal Tap Logic Analyzer that is included with the following software:

- Intel Quartus Prime design software
- Intel Quartus Prime Lite Edition

Alternatively, use the Signal Tap Logic Analyzer standalone software and standalone Programmer software.

Note: The Intel Quartus Prime Lite Edition software does not support incremental compilation integration with the Signal Tap Logic Analyzer.

The memory blocks of the device store captured data. The memory blocks transfer the data to the Intel Quartus Prime software waveform display over a JTAG communication cable, such as or Intel FPGA Download Cable.

Table 109. Signal Tap Logic Analyzer Features and Benefits

Feature	Benefit
Quick access toolbar	Provides single-click operation of commonly-used menu items. You can hover over the icons to see tool tips.
Multiple logic analyzers in a single device	Allows you to capture data from multiple clock domains in a design at the same time.
Multiple logic analyzers in multiple devices in a single JTAG chain	Allows you to capture data simultaneously from multiple devices in a JTAG chain.
Nios II plug-in support	Allows you to specify nodes, triggers, and signal mnemonics for IP, such as the Nios II processor.
Up to 10 basic, comparison, or advanced trigger conditions for each analyzer instance	Allows you to send complex data capture commands to the logic analyzer, providing greater accuracy and problem isolation.
Power-up trigger	Captures signal data for triggers that occur after device programming, but before manually starting the logic analyzer.
Custom trigger HDL object	You can code your own trigger in Verilog HDL or VHDL and tap specific instances of modules located anywhere in the hierarchy of your design, without needing to manually route all the necessary connections. This simplifies the process of tapping nodes spread out across your design.
State-based triggering flow	Enables you to organize your triggering conditions to precisely define what your logic analyzer captures.
Incremental compilation	Allows you to modify the signals and triggers that the Signal Tap Logic Analyzer monitors without performing a full compilation, saving time.
Incremental route with rapid recompile	Allows you to manually allocate trigger input, data input, storage qualifier input, and node count, and perform a full compilation to include the Signal Tap Logic Analyzer in your design. Then, you can selectively connect, disconnect, and swap to different nodes in your design. Use Rapid Recompile to perform incremental routing and gain a 2-4x speedup over the initial full compilation.
Flexible buffer acquisition modes	The buffer acquisition control allows you to precisely control the data that is written into the acquisition buffer. Both segmented buffers and non-segmented buffers with storage qualification allow you to discard data samples that are not relevant to the debugging of your design.
MATLAB integration with included MEX function	Collects the data the Signal Tap Logic Analyzer captures into a MATLAB integer matrix.
Up to 2,048 channels per logic analyzer instance	Samples many signals and wide bus structures.
Up to 128K samples per instance	Captures a large sample set for each channel.
Fast clock frequencies	Synchronous sampling of data nodes using the same clock tree driving the logic under test.
Resource usage estimator	Provides an estimate of logic and memory device resources that the Signal Tap Logic Analyzer configurations use.

continued...

Feature	Benefit
No additional cost	Intel Quartus Prime subscription and the Intel Quartus Prime Lite Edition include the Signal Tap Logic Analyzer.
Compatibility with other on-chip debugging utilities	You can use the Signal Tap Logic Analyzer in tandem with any JTAG-based on-chip debugging tool, such as an In-System Memory Content editor, allowing you to change signal values in real-time while you are running an analysis with the Signal Tap Logic Analyzer.
Floating-Point Display Format	To enable, click **Edit ➤ Bus Display Format ➤ Floating-point** Supports: • Single-precision floating-point format **IEEE754 Single (32-bit)**. • Double-precision floating-point format **IEEE754 Double (64-bit)**.

Related Links

System Debugging Tools Overview on page 183

14.1.2 Open Standalone Signal Tap Logic Analyzer GUI

To open a new Signal Tap through the command-line, type:

```
quartus_stpw <stp_file.stp>
```

14.1.3 Backward Compatibility with Previous Versions of Intel Quartus Prime Software

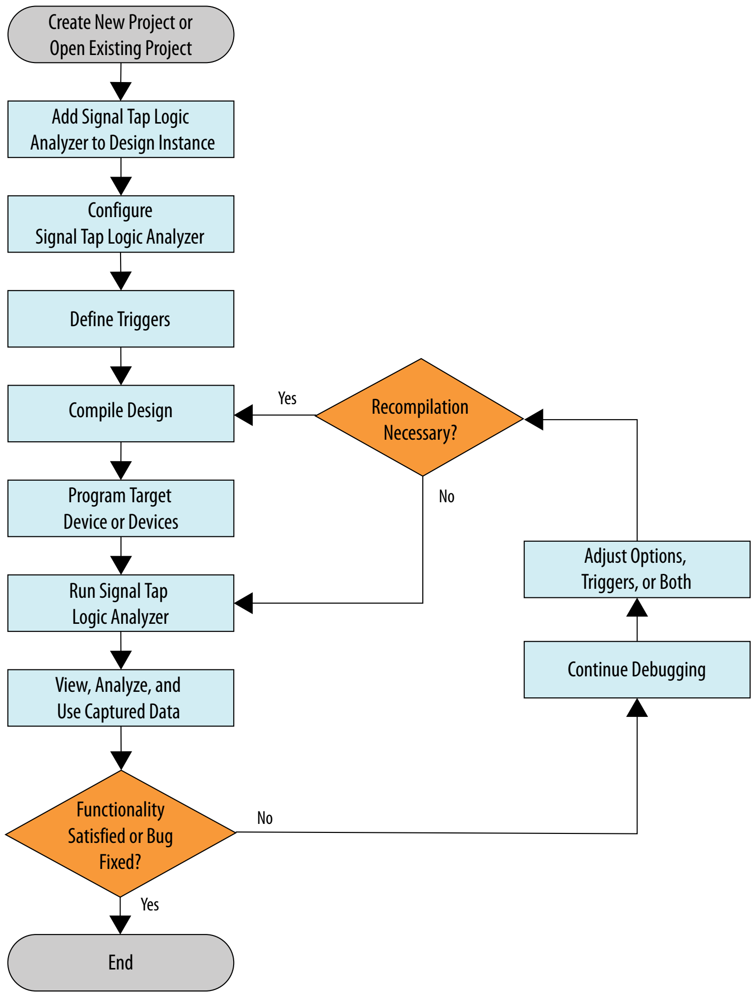
When you open an `.stp` file created in a previous version of Intel Quartus Prime software in a newer version of the software, the `.stp` file cannot be opened in a previous version of the Intel Quartus Prime software.

If you have a Intel Quartus Prime project file from a previous version of the software, you may have to update the `.stp` configuration file to recompile the project. You can update the configuration file by opening the Signal Tap Logic Analyzer. If you need to update your configuration, a prompt appears asking if you want to update the `.stp` to match the current version of the Intel Quartus Prime software.

14.2 Signal Tap Logic Analyzer Task Flow Overview

To use the Signal Tap Logic Analyzer to debug your design, you perform a number of tasks to add, configure, and run the logic analyzer.

Figure 162. Signal Tap Logic Analyzer Task Flow


Create New Project or Open Existing Project
Add Signal Tap Logic Analyzer to Design Instance
Configure Signal Tap Logic Analyzer
Define Triggers
Compile Design
Yes
Recompilation Necessary?
No
Program Target Device or Devices
Run Signal Tap Logic Analyzer
Adjust Options, Triggers, or Both
Continue Debugging
View, Analyze, and Use Captured Data
Functionality Satisfied or Bug Fixed?
No
Yes
End

14.2.1 Add the Signal Tap Logic Analyzer to Your Design

Create an `.stp` or create a parameterized HDL instance representation of the logic analyzer using the IP Catalog and parameter editor. If you want to monitor multiple clock domains simultaneously, add additional instances of the logic analyzer to your design, limited only by the available resources in your device.

14.2.2 Configure the Signal Tap Logic Analyzer

After you add the Signal Tap Logic Analyzer to your design, configure the logic analyzer to monitor the signals you want.

You can add signals manually or use a plug-in, such as the Nios II processor plug-in, to add entire sets of associated signals for a particular IP.

Specify settings for the data capture buffer, such as its size, the method in which the Signal Tap Logic Analyzer captures and stores the data. If your device supports memory type selection, you can specify the memory type to use for the buffer.

Related Links

Configuring the Signal Tap Logic Analyzer on page 333

14.2.3 Define Trigger Conditions

To capture and store specific signal data, set up triggers that tell the logic analyzer under what conditions to stop capturing data. The Signal Tap Logic Analyzer captures data continuously while the logic analyzer is running.

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

Related Links

Defining Triggers on page 352

14.2.4 Compile the Design

Once you configure the `.stp` file and define trigger conditions, compile your project including the logic analyzer in your design.

Note: Because you may need to change monitored signal nodes or adjust trigger settings frequently during debugging, Intel FPGA recommends that you use the incremental compilation feature built into the Signal Tap Logic Analyzer, along with Intel Quartus Prime incremental compilation, to reduce recompile times. You can also use Incremental Route with Rapid Recompile to reduce recompile times.

Related Links

Compiling the Design on page 376

14.2.5 Program the Target Device or Devices

When you debug a design with the Signal Tap Logic Analyzer, you can program a target device directly from the `.stp` without using the Intel Quartus Prime Programmer. You can also program multiple devices with different designs and simultaneously debug them.

Related Links

- Program the Target Device or Devices on page 381
- Manage Multiple Signal Tap Files and Configurations on page 350

14.2.6 Run the Signal Tap Logic Analyzer

In normal device operation, you control the logic analyzer through the JTAG connection, specifying when to start looking for trigger conditions to begin capturing data. With Runtime or Power-Up Triggers, read and transfer the captured data from the on-chip buffer to the `.stp` for analysis.

Related Links

Running the Signal Tap Logic Analyzer on page 382

14.2.7 View, Analyze, and Use Captured Data

The data you capture and read into the `.stp` file is available for analysis and debugging. You can save the data for later analysis, or convert the data to other formats for sharing and further study.

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

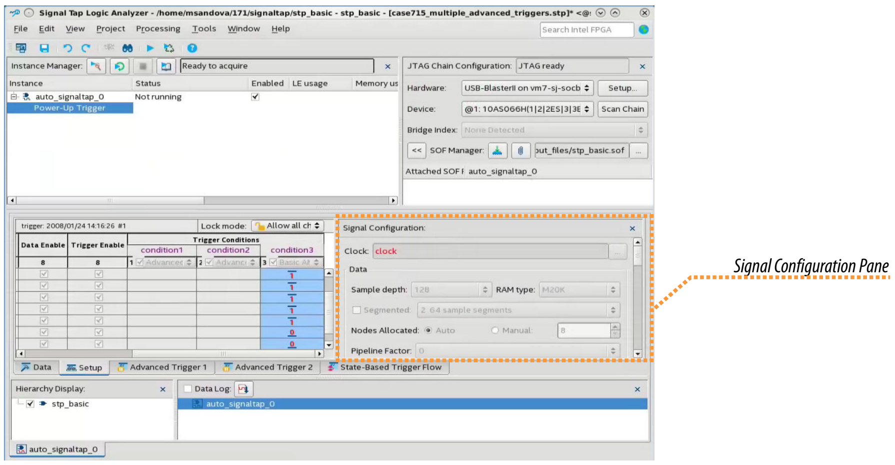
Related Links

View, Analyze, and Use Captured Data on page 386

14.3 Configuring the Signal Tap Logic Analyzer

You can configure instances of the Signal Tap Logic Analyzer in the **Signal Configuration** pane of the **Signal Tap Logic Analyzer** window. Some settings are similar to those found on traditional external logic analyzers. Other settings are unique to the Signal Tap Logic Analyzer.

Figure 163. Signal Tap Logic Analyzer Signal Configuration Pane



Note: You can adjust fewer settings with run-time trigger conditions than with power-up trigger conditions.

14.3.1 Assigning an Acquisition Clock

To control how the Signal Tap Logic Analyzer acquires data you must assign a clock signal. The logic analyzer samples data on every positive (rising) edge of the acquisition clock. The logic analyzer does not support sampling on the negative (falling) edge of the acquisition clock.

You can use any signal in your design as the acquisition clock. However, for best results in data acquisition, use a global, non-gated clock that is synchronous to the signals under test. Using a gated clock as your acquisition clock can result in unexpected data that does not accurately reflect the behavior of your design. The Intel

Quartus Prime static timing analysis tools show the maximum acquisition clock frequency at which you can run your design. To find the maximum frequency of the logic analyzer clock, refer to the Timing Analysis section of the Compilation Report.

Caution: Be careful when using a recovered clock from a transceiver as an acquisition clock for the Signal Tap Logic Analyzer. A recovered clock can cause incorrect or unexpected behavior, particularly when the transceiver recovered clock is the acquisition clock with the power-up trigger feature.

If you do not assign an acquisition clock in the Signal Tap Logic Analyzer Editor, Intel Quartus Prime software automatically creates a clock pin called `auto_stp_external_clk`. You must make a pin assignment to this pin, and make sure that a clock signal in your design drives the acquisition clock.

Related Links

- Adding Signals with a Plug-In on page 337
- Managing Device I/O Pins
  In *Intel Quartus Prime Standard Edition Handbook Volume 2*

14.3.2 Adding Signals to the Signal Tap File

Add the signals that you want to monitor to the `.stp` node list. You can also select signals to define triggers. You can assign the following two signal types:

- **Pre-synthesis**—These signals exist after design elaboration, but before any synthesis optimizations are done. This set of signals must reflect your Register Transfer Level (RTL) signals.
- **Post-fitting**—These signals exist after physical synthesis optimizations and place-and-route.

Note: If you are not using incremental compilation, add only pre-synthesis signals to the `.stp`. Using pre-synthesis helps when you want to add a new node after you change a design. After you perform Analysis and Elaboration, the source file changes appear in the Node Finder.

Intel Quartus Prime software does not limit the number of signals available for monitoring in the Signal Tap window waveform display. However, the number of channels available is directly proportional to the number of logic elements (LEs) or adaptive logic modules (ALMs) in the device. Therefore, there is a physical restriction on the number of channels that are available for monitoring. Signals shown in blue text are post-fit node names. Signals shown in black text are pre-synthesis node names.

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

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

Related Links

- Faster Compilations with Intel Quartus Prime Incremental Compilation on page 376
- Setup Tab (Signal Tap Logic Analyzer)
  In *Intel Quartus Prime Help*

14.3.2.1 About Adding Pre-Synthesis Signals

When you add pre-synthesis signals, make all connections to the Signal Tap Logic Analyzer before synthesis. The Compiler allocates logic and routing resources to make the connection as if you changed your design files. For signals driving to and from IOEs, pre-synthesis signal names coincide with the pin's signal names.

14.3.2.2 About Adding Post-Fit Signals

In the case of post-fit signals, connections that you make to the Signal Tap Logic Analyzer are the signal names from the actual atoms in your post-fit netlist. You can only make a connection if the signals are part of the existing post-fit netlist, and existing routing resources are available from the signal of interest to the Signal Tap Logic Analyzer.

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

Note: Because `NOT`-gate push back applies to any register that you tap, the signal from the atom may be inverted. You can check this by locating the signal in either the Resource Property Editor or the Technology Map Viewer. You can also use the Technology Map viewer and the Resource Property Editor to find post-fit node names.

Related Links

Design Flow with the Netlist Viewers
In *Intel Quartus Prime Standard Edition Handbook Volume 1*

14.3.2.2.1 Assigning Data Signals Using the Technology Map Viewer

You can use the Technology Map Viewer to add post-fit signal names easily. To do so, launch the Technology Map Viewer (post-fitting) after compilation. When you find the desired node, copy the node to either the active `.stp` for your design or a new `.stp`.

To launch the Technology Map Viewer, click **Tools ➤ Netlist Viewers ➤ Technology Map Viewer (Post-Fitting)** in the **Intel Quartus Prime** window.

14.3.2.3 Preserving Signals

The Intel Quartus Prime software optimizes the RTL signals during synthesis and place-and-route. RTL signal names may not appear in the post-fit netlist after optimizations. For example, the compilation process can add tildes (~) to nets that fan-out from a node, making it difficult to decipher which signal nets they actually represent.

The Intel Quartus Prime software provides synthesis attributes that prevent the Compiler to perform any optimization on the specified signals, allowing them to persist into the post-fit netlist:

- `keep`—Prevents removal of combinational signals during optimization.
- `preserve`—Prevents removal of registers during optimization.

However, using preserving attributes can increase device resource utilization or decrease timing performance.

Note: These processing results can cause problems when you use the incremental compilation flow with the Signal Tap Logic Analyzer. Because you can only add post-fitting signals to the Signal Tap Logic Analyzer in partitions of type **post-fit**, RTL signals that you want to monitor may not be available, preventing their use. To avoid this issue, use synthesis attributes to preserve signals during synthesis and place-and-route.

If you are debugging an IP core, such as the Nios II CPU or other encrypted IP, you might need to preserve nodes from the core to keep available for debugging with the Signal Tap Logic Analyzer. Preserving nodes is often necessary when you use a plug-in to add a group of signals for a particular IP.

If you use incremental compilation flow with the Signal Tap Logic Analyzer, pre-synthesis nodes may not be connected to the Signal Tap Logic Analyzer if the affected partition is of the post-fit type. Signal Tap issues a critical warning for all pre-synthesis node names that it does not find in the post-fit netlist.

14.3.2.4 Node List Signal Use Options

When you add a signal to the node list, you can select options that specify how the logic analyzer uses the signal.

To prevent a signal from triggering the analysis, disable the signal's **Trigger Enable** option in the `.stp` file. This option is useful when you only want to see the signal's captured data.

You can turn off the ability to view data for a signal by disabling the **Data Enable** column in the `.stp` file. This option is useful when you want to trigger on a signal, but have no interest in viewing that signal's data.

Related Links

Defining Triggers on page 352

14.3.2.4.1 Disabling and Enabling a Signal Tap Instance

Disable and enable Signal Tap instances in the **Instance Manager** pane. Physically adding or removing instances requires recompilation after disabling and enabling a Signal Tap instance.

14.3.2.5 Untappable Signals

Not all the post-fitting signals in your design are available in the **Signal Tap : post-fitting filter** in the **Node Finder** dialog box.

You cannot tap any of the following signal types:

- **Post-fit output pins**—You cannot tap a post-fit output pin directly. To make an output signal visible, tap the register or buffer that drives the output pin. This includes pins defined as bidirectional.
- **Signals that are part of a carry chain**—You cannot tap the carry out (`cout0` or `cout1`) signal of a logic element. Due to architectural restrictions, the carry out signal can only feed the carry in of another LE.
- **JTAG Signals**—You cannot tap the JTAG control (`TCK`, `TDI`, `TDO`, and `TMS`) signals.
- **ALTGXB IP core**—You cannot directly tap any ports of an ALTGXB instantiation.
- **LVDS**—You cannot tap the data output from a serializer/deserializer (SERDES) block.
- **DQ**, **DQS Signals**—You cannot directly tap the `DQ` or `DQS` signals in a DDR/DDRII design.

14.3.3 Adding Signals with a Plug-In

Instead of adding individual or grouped signals through the **Node Finder**, you can use a plug-in to add groups of relevant signals of a particular type of IP. Besides easy signal addition, plug-ins provide features such as pre-designed mnemonic tables, useful for trigger creation and data viewing, as well as the ability to disassemble code in captured data. The Signal Tap Logic Analyzer comes with one plug-in for the Nios II processor.

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

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

To add signals to the `.stp` file using a plug-in, perform the following steps after running Analysis and Elaboration on your design:

1. To ensure that all the required signals are available, in the Intel Quartus Prime software, click **Assignments ➤ Settings ➤ Compiler Settings ➤ Advanced Settings (Synthesis)**. Turn on **Create debugging nodes for IP cores**.
   All the signals included in the plug-in are added to the node list.
2. Right-click the node list. On the **Add Nodes with Plug-In** submenu, select the plug-in you want to use, such as the included plug-in named **Nios II**.
   The **Select Hierarchy Level** dialog box appears showing the IP hierarchy of your design. If the IP for the selected plug-in does not exist in your design, a message informs you that you cannot use the selected plug-in.
3. Select the IP that contains the signals you want to monitor with the plug-in, and click **OK**.

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

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

Related Links

- Defining Triggers on page 352
- View, Analyze, and Use Captured Data on page 333

14.3.4 Adding Finite State Machine State Encoding Registers

Finding the signals to debug finite state machines (FSM) can be challenging. Finding nodes from the post-fit netlist may be impossible, since the Compiler may change or optimize away FSM encoding signals. To find and map FSM signal values to the state names that you specified in your HDL, you must perform an additional step.

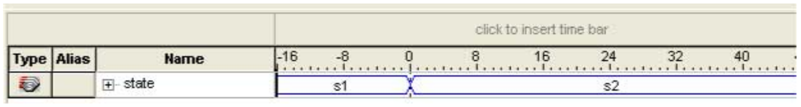
The Signal Tap Logic Analyzer can detect FSMs in your compiled design. The configuration automatically tracks the FSM state signals as well as state encoding through the compilation process.

To add all the FSM state signals to your logic analyzer with a single command Shortcut menu commands allow you .

For each FSM added to your Signal Tap configuration, the FSM debugging feature adds a mnemonic table to map the signal values to the state enumeration that you provided in your source code. The mnemonic tables enable you to visualize state machine transitions in the waveform viewer. The FSM debugging feature supports adding FSM signals from both the pre-synthesis and post-fit netlists.

Figure 164. Decoded FSM Mnemonics

The waveform viewer with decoded signal values from a state machine added with the FSM debugging feature.



Related Links

State Machine HDL Guidelines
In *Intel Quartus Prime Standard Edition Handbook Volume 1*

14.3.4.1 Modify and Restore Mnemonic Tables for State Machines

Edit any mnemonic table using the **Mnemonic Table Setup** dialog box. When you add FSM state signals via the FSM debugging feature, the Signal Tap Logic Analyzer GUI creates a mnemonic table using the format <*StateSignalName*>`_table`, where **StateSignalName** is the name of the state signals that you have declared in your RTL.

If you want to restore a mnemonic table that was modified, right-click anywhere in the node list window and select **Recreate State Machine Mnemonics**. By default, restoring a mnemonic table overwrites the existing mnemonic table that you modified. To restore a FSM mnemonic table to a new record, turn off **Overwrite existing mnemonic table** in the **Recreate State Machine Mnemonics** dialog box.

Note: If you have added or deleted a signal from the FSM state signal group from within the setup tab, delete the modified register group and add the FSM signals back again.

Related Links

Creating Mnemonics for Bit Patterns on page 389

14.3.4.2 Additional Considerations for State Machines in Signal Tap

- The Signal Tap configuration GUI recognizes state machines from your design only if you use Intel Quartus Prime Integrated Synthesis. Conversely, the state machine debugging feature is not able to track the FSM signals or state encoding if you use other EDA synthesis tools.
- If you add post-fit FSM signals, the Signal Tap Logic Analyzer FSM debug feature may not track all optimization changes that are a part of the compilation process.
- If the following two specific optimizations are enabled, the Signal Tap FSM debug feature may not list mnemonic tables for state machines in the design:
  - If you enabled the **Physical Synthesis** optimization, state registers may be resource balanced (register retiming) to improve f_{MAX}. The FSM debug feature does not list post-fit FSM state registers if register retiming occurs.
  - The FSM debugging feature does not list state signals that the Compiler packed into RAM and DSP blocks during synthesis or Fitter optimizations.
- You can still use the FSM debugging feature to add pre-synthesis state signals.

Related Links

Enabling Physical Synthesis Optimization
In *Intel Quartus Prime Standard Edition Handbook Volume 1*

14.3.5 Specify the Sample Depth

The **Sample depth** setting specifies the number of samples the Signal Tap Logic Analyzer captures and stores, for each signal in the captured data buffer. To specify the sample depth, select the desired number in the **Sample Depth** drop-down menu. The sample depth ranges from 0 to 128K.

If device memory resources are limited, you may not be able to successfully compile your design with the sample buffer size you have selected. Try reducing the sample depth to reduce resource usage.

Related Links

Signal Configuration Pane (View Menu) (Signal Tap Logic Analyzer)
In *Intel Quartus Prime Help*

14.3.6 Capture Data to a Specific RAM Type

You have the option to select the RAM type where the Signal Tap Logic Analyzer stores acquisition data. Once you allocate the Signal Tap Logic Analyzer buffer to a particular RAM block, the entire RAM block becomes a dedicated resource for the logic analyzer.

RAM selection allows you to preserve a specific memory block for your design, and allocate another portion of memory for Signal Tap Logic Analyzer data acquisition.

For example, if your design has an application that requires a large block of memory resources, such as a large instruction or data cache, you can use MLAB, M512, or M4k blocks for data acquisition and leave M9k blocks for the rest of your design.

To specify the RAM type to use for the Signal Tap Logic Analyzer buffer, go to the **Signal Configuration** pane in the **Signal Tap** window, and select one **Ram type** from the drop-down menu.

Use this feature only when the acquired data is smaller than the available memory of the RAM type that you selected. The amount of data appears in the Signal Tap resource estimator.

Related Links

Signal Configuration Pane (View Menu) (Signal Tap Logic Analyzer)
In *Intel Quartus Prime Help*

14.3.7 Select the Buffer Acquisition Mode

When you specify how the logic analyzer organizes the captured data buffer, you can potentially reduce the amount of memory that Signal Tap requires for data acquisition.

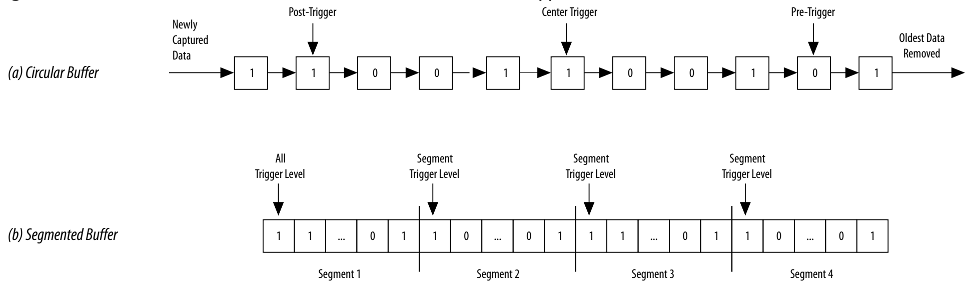
There are two types of acquisition buffer within the Signal Tap Logic Analyzer—a non-segmented (or circular) buffer and a segmented buffer.

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

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

Figure 165. Buffer Type Comparison in the Signal Tap Logic Analyzer

The figure illustrates the differences between the two buffer types.



Both non-segmented and segmented buffers can use a preset trigger position (Pre-Trigger, Center Trigger, Post-Trigger). Alternatively, you can define a custom trigger position using the **State-Based Triggering** tab. Refer to *Specify Trigger Position* for more details.

Notes to figure:

Related Links

- Specify Trigger Position on page 372
- Using the Storage Qualifier Feature on page 343

14.3.7.1 Non-Segmented Buffer

The non-segmented buffer is the default buffer type in the Signal Tap Logic Analyzer.

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

- To capture most data before the trigger occurs, select **Post trigger position** from the list
- To capture most data after the trigger, select **Pre trigger position**.
- To center the trigger position in the data, select **Center trigger position**.

Alternatively, use the custom State-based triggering flow to define a custom trigger position within the capture buffer.

Related Links

Specify Trigger Position on page 372

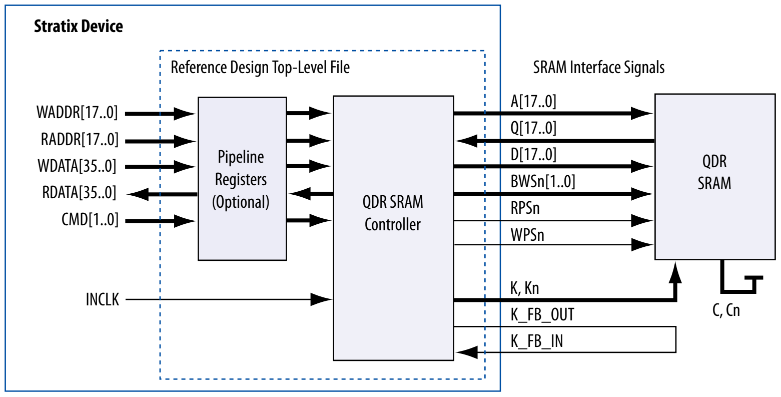
14.3.7.2 Segmented Buffer

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

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

Figure 166. System that Generates Recurring Events

In this design, you want to ensure that the correct data is written to the SRAM controller by monitoring the RDATA port whenever the address `H'0F0F0F0F` is sent into the RADDR port.



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

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

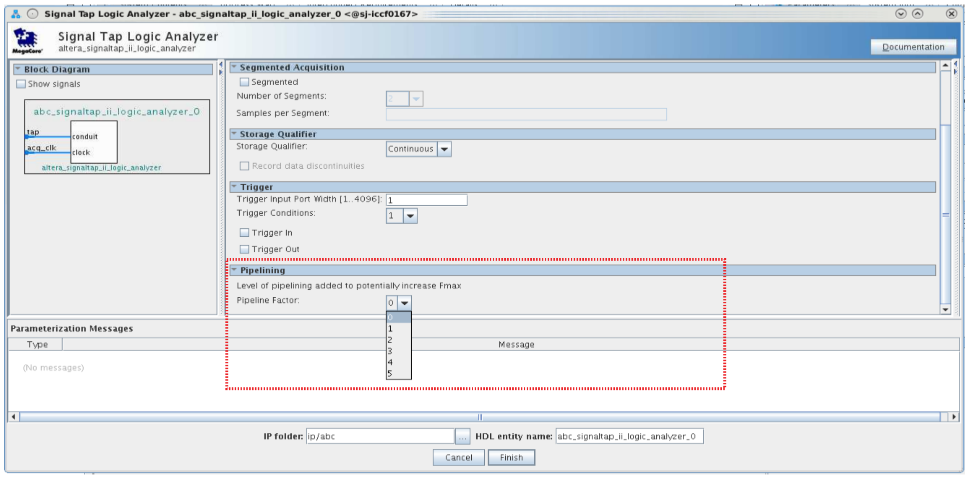
14.3.8 Specify the Pipeline Factor

The **Pipeline factor** setting indicates the number of pipeline registers that you can add to boost the f_{MAX} of the Signal Tap Logic Analyzer. You can specify the pipeline factor in the **Signal Configuration** pane. The pipeline factor ranges from 0 to 5, with a default value of 0.

You can also set the pipeline factor when you instantiate the Signal Tap Logic Analyzer component from your Platform Designer (Standard) system:

1. Double-click **Signal Tap Logic Analyzer** component in the IP Catalog.
2. Specify the **Pipeline Factor**, along with other parameter values.

Figure 167. Specifying the Pipeline Factor from Platform Designer (Standard)



Signal Tap Logic Analyzer - abc_signaltap_ii_logic_analyzer_0 <@sj-iccf0167>

Signal Tap Logic Analyzer
altera_signaltap_ii_logic_analyzer

Block Diagram
☐ Show signals

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

Parameterization Messages
Type | Message
(No messages)

IP folder: ip/abc
HDL entity name: abc_signaltap_ii_logic_analyzer_0

Documentation
abc_signaltap_ii_logic_analyzer_0
tap
acq_clk
conduit
clock
altera_signaltap_ii_logic_analyzer
0
1
2
3
4
5
Cancel
Finish

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

14.3.9 Using the Storage Qualifier Feature

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

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

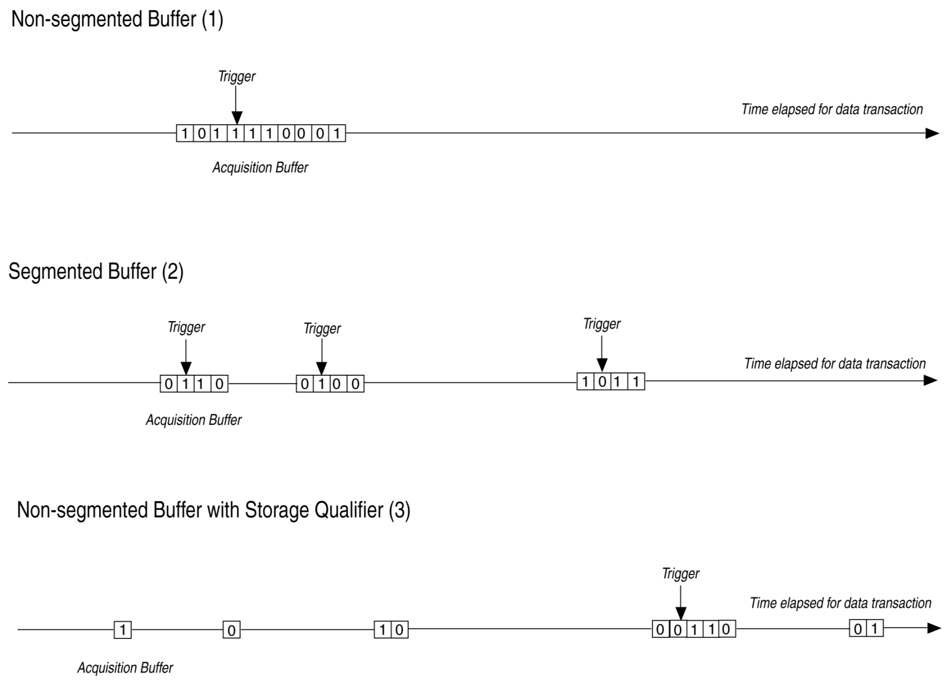
With analysis using acquisition buffers you can capture most functional errors in a chosen signal set, provided adequate trigger conditions and a generous sample depth for the acquisition. However, each data window can have a considerable amount of unnecessary data; for example, long periods of idle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

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

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

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

Figure 168. Data Acquisition Using Different Modes of Controlling the Acquisition Buffer



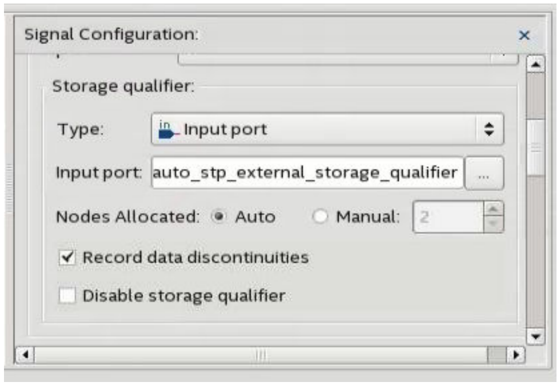
Notes to figure:

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

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

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

Figure 169. Storage Qualifier Settings


Signal Configuration:

Storage qualifier:

Type: Input port

Input port: auto_stp_external_storage_qualifier

Nodes Allocated: (● Auto) ○ Manual: 2

[✓] Record data discontinuities

[ ] Disable storage qualifier

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

The Signal Tap Logic Analyzer evaluates trigger conditions independently of storage qualifier conditions.

Related Links

Define Trigger Conditions on page 332

14.3.9.1 Input Port Mode

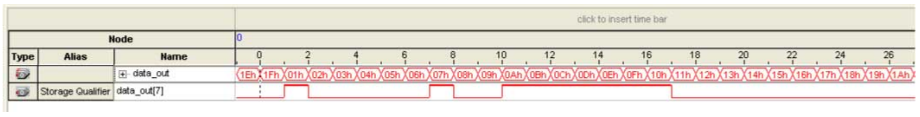
When using the Input port mode, the Signal Tap Logic Analyzer takes any signal from your design as an input. During acquisition, if the signal is high on the clock edge, the Signal Tap Logic Analyzer stores the data in the buffer. If the signal is low on the clock edge, the Logic Analyzer ignores the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

If you are creating a Signal Tap Logic Analyzer instance through an `.stp` file, specify the storage qualifier signal using the input port field located on the **Setup** tab. You must specify this port for your project to compile.

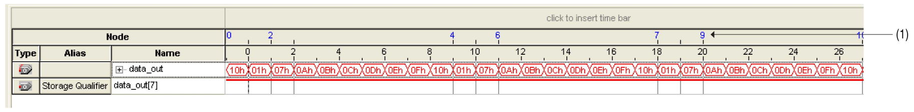
If you use the parameter editor, the storage qualification input port, if specified, appears in the generated instantiation template. You can then connect this port to a signal in your RTL.

Figure 170. Comparing Continuous and Input Port Capture Mode in Data Acquisition of a Recurring Data Pattern

- Continuous Mode:


click to insert time bar
Node
Type
Alias
Name
data_out
Storage Qualifier
data_out[7]
1Eh 1Fh 01h 02h 03h 04h 05h 06h 07h 08h 09h 0Ah 0Bh 0Ch 0Dh 0Eh 0Fh 10h 11h 12h 13h 14h 15h 16h 17h 18h 19h 1Ah

- Input Port Storage Qualifier:

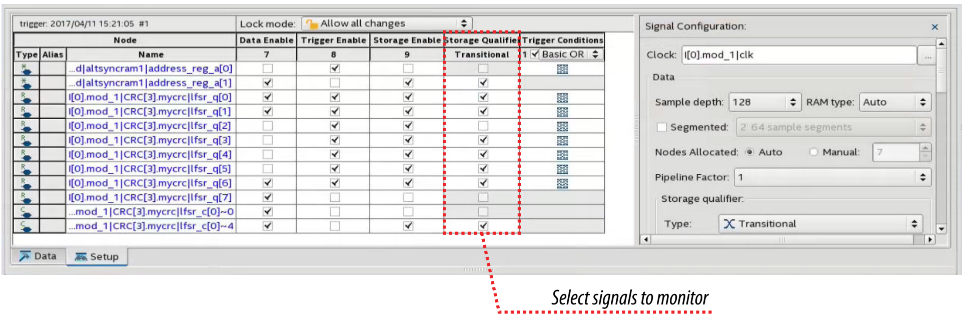

click to insert time bar
Node
Type
Alias
Name
data_out
Storage Qualifier
data_out[7]
(1)
10h 01h 07h 0Ah 0Bh 0Ch 0Dh 0Eh 0Fh 10h 01h 07h 0Ah 0Bh 0Ch 0Dh 0Eh 0Fh 10h 01h 07h 0Ah 0Bh 0Ch 0Dh 0Eh 0Fh 10h

(1) Markers display samples when the logic analyzer paused a write into acquisition memory. These markers are enabled with the option "Record data discontinuities."

14.3.9.2 Transitional Mode

In Transitional mode, the Logic Analyzer monitors changes in a set of signals, and writes new data in the acquisition buffer only when it detects a change. You select the signals for monitoring using the check boxes in the **Storage Qualifier** column.

Figure 171. Transitional Storage Qualifier Setup



trigger: 2017/04/11 15:21:05 #1

Lock mode: Allow all changes

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	✓		✓	✓	

Data | Setup

Signal Configuration:

Clock: I[0].mod_1|clk

Data

Sample depth: 128 RAM type: Auto

Segmented: 2 64 sample segments

Nodes Allocated: [✓ Auto] Manual: 7

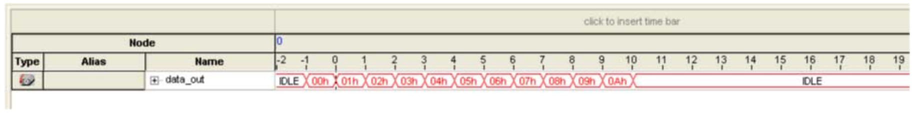
Pipeline Factor: 1

Storage qualifier:

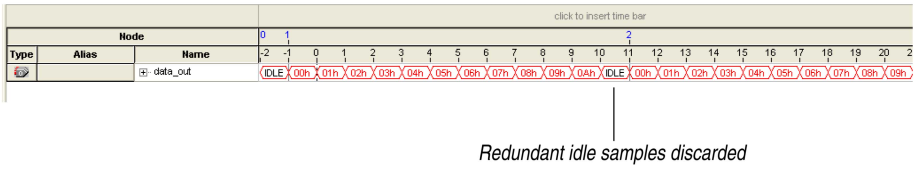
Type: Transitional

Figure 172. Comparing Continuous and Transitional Capture Mode in Data Acquisition of a Recurring Data Pattern

- Continuous:



- Transitional mode:


click to insert time bar
Node
Type
Alias
Name
data_out
IDLE 00h 01h 02h 03h 04h 05h 06h 07h 08h 09h 0Ah IDLE 00h 01h 02h 03h 04h 05h 06h 07h 08h 09h
Redundant idle samples discarded

14.3.9.3 Conditional Mode

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

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

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

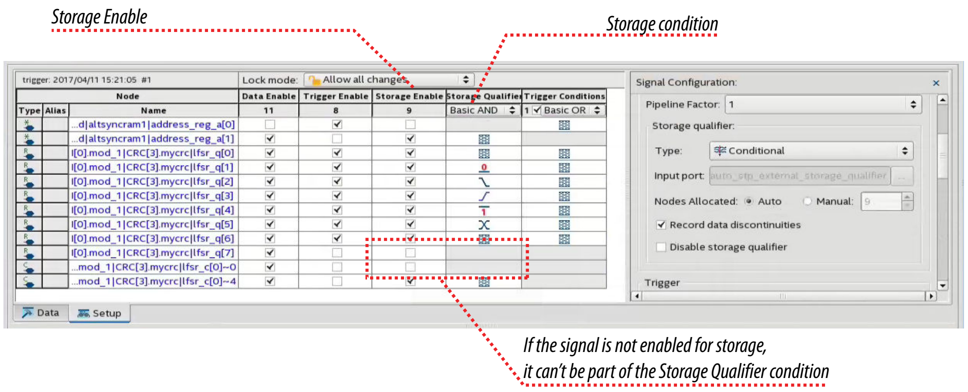
If you specify a **Basic AND** storage qualifier condition for more than one signal, the Signal Tap Logic Analyzer evaluates the logical AND of the conditions.

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

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

Figure 173. Conditional Storage Qualifier Setup

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


Storage Enable
Storage condition
If the signal is not enabled for storage,
it can't be part of the Storage Qualifier condition
Data
Setup

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

Node			Data Enable	Trigger Enable	Storage Enable	Storage Qualifier	Trigger Conditions
Type	Alias	Name	11	8	9	Basic AND	1 ✓ Basic OR
		...d|altsyncram1|address_reg_a[0]	☐	✓	☐		▨
		...d|altsyncram1|address_reg_a[1]	✓	☐	✓	▨	
		I[0].mod_1|CRC[3].mycrc|lfsr_q[0]	✓	✓	✓	▨	▨
		I[0].mod_1|CRC[3].mycrc|lfsr_q[1]	✓	✓	✓	0	▨
		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]	✓	✓	✓	1	▨
		I[0].mod_1|CRC[3].mycrc|lfsr_q[5]	✓	✓	✓	X	▨
		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:

Pipeline Factor: 1

Storage qualifier:

Type: Conditional

Input port: auto_stp_external_storage_qualifier

Nodes Allocated: [✓ Auto] Manual: 9

✓ Record data discontinuities

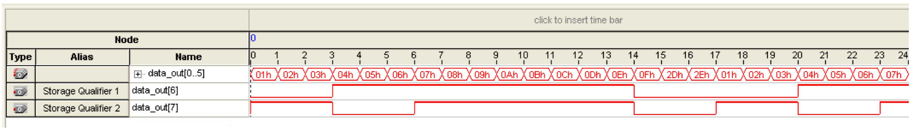
☐ Disable storage qualifier

Trigger

Figure 174. Comparing Continuous and Conditional Capture Mode in Data Acquisition of a Recurring Data Pattern

The data pattern is the same in both cases.

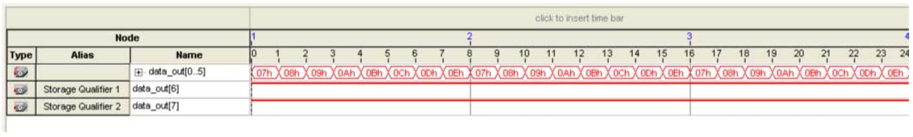
- Continuous sampling capture mode:



```
                                         click to insert time bar
Node                                  0
Type  Alias                Name       0    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19   20   21   22   23   24
      ⊞ data_out[0..5]                01h  02h  03h  04h  05h  06h  07h  08h  09h  0Ah  0Bh  0Ch  0Dh  0Eh  0Fh  2Dh  2Eh  01h  02h  03h  04h  05h  06h  07h
      Storage Qualifier 1  data_out[6]
      Storage Qualifier 2  data_out[7]
```

(1) Storage Qualifier condition is set up to evaluate data_out[6] AND data_out[7].

- Conditional sampling capture mode:



```
                                         click to insert time bar
Node                                  1                                       2                                       3                                       4
Type  Alias                Name       0    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19   20   21   22   23   24
      ⊞ data_out[0..5]                07h  08h  09h  0Ah  0Bh  0Ch  0Dh  0Eh  07h  08h  09h  0Ah  0Bh  0Ch  0Dh  0Eh  07h  08h  09h  0Ah  0Bh  0Ch  0Dh  0Eh
      Storage Qualifier 1  data_out[6]
      Storage Qualifier 2  data_out[7]
```

Related Links

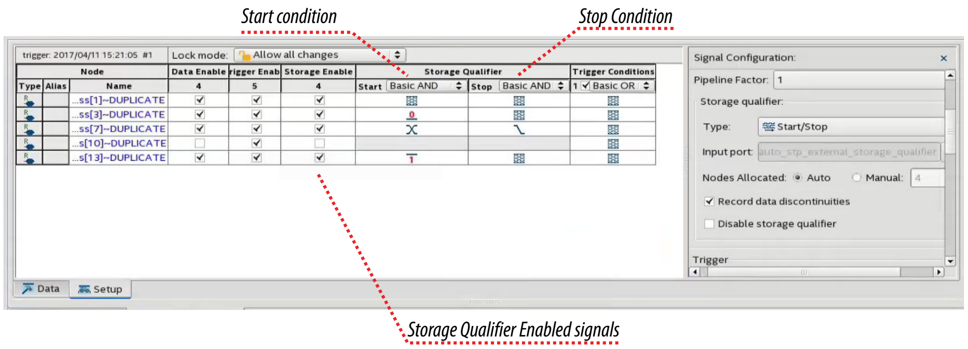
- Basic Trigger Conditions on page 352
- Comparison Trigger Conditions on page 353
- Advanced Trigger Conditions on page 355

14.3.9.4 Start/Stop Mode

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

Note: You can force a trigger by pressing the **Stop** button if the buffer fails to fill to completion due to a stop condition.

Figure 175. Start/Stop Mode Storage Qualifier Setup



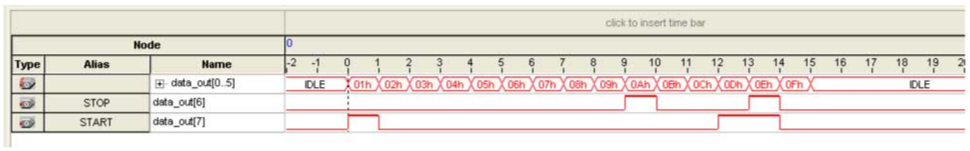
trigger: 2017/04/11 15:21:05 #1   Lock mode: Allow all changes

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

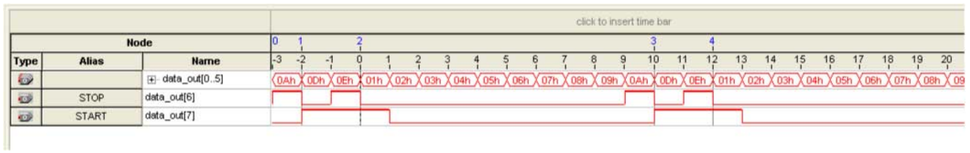
Signal Configuration:
Pipeline Factor: 1
Storage qualifier:
Type: Start/Stop
Input port: auto_stp_external_storage_qualifier
Nodes Allocated: [✓ Auto] Manual: 4
[✓] Record data discontinuities
[ ] Disable storage qualifier
Trigger

Figure 176. Comparing Continuous and Start/Stop Acquisition Modes for a Recurring Data Pattern

- Continuous Mode:



- Start/Stop Storage Qualifier:



14.3.9.5 State-Based

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

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

Related Links

State-Based Triggering on page 363

14.3.9.6 Showing Data Discontinuities

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

14.3.9.7 Disable Storage Qualifier

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

Related Links

Runtime Reconfigurable Options on page 383

14.3.10 Manage Multiple Signal Tap Files and Configurations

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

Signal Tap Logic Analyzer allows you to save debug configurations in more than one `.stp` file. Alternatively, you can embed multiple configurations within the same `.stp` file, and use the Data Log as a managing tool.

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

Related Links

Ensure Setting Compatibility Between .stp and .sof Files on page 382

14.3.10.1 Data Log Pane

The Data Log pane displays all Signal Tap configurations and data capture results stored within a single `.stp` file.

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

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

Table 110. Data Log Items

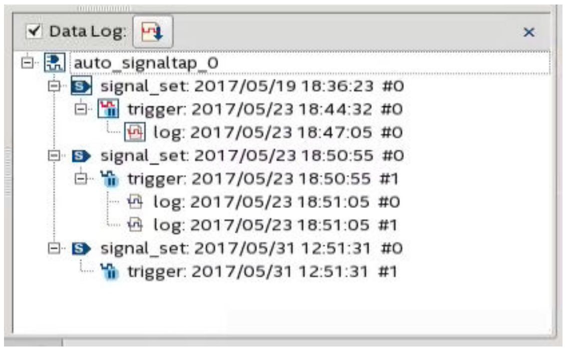
Item	Icon		Contains one or more	Comments
	Unselected	Selected		
Instance			Signal Set	
Signal Set			Trigger	The Signal Set changes whenever you add a new signal to Signal Tap. After a change in the Signal Set, you need to recompile.
Trigger			Capture Log	A trigger changes when you change any trigger condition. These changes do not require recompilation.
Capture Log				

The name on each entry displays the wall-clock time when Signal Tap Logic Analyzer triggered, and the time elapsed from start acquisition to trigger activation. You can rename entries so they make sense to you.

To switch between configurations, double-click an entry in the Data Log. As a result, the **Setup** tab updates to display the active signal list and trigger conditions.

Example 34. Simple Data Log

On this example, the Data Log displays one instance with three signal set configurations.



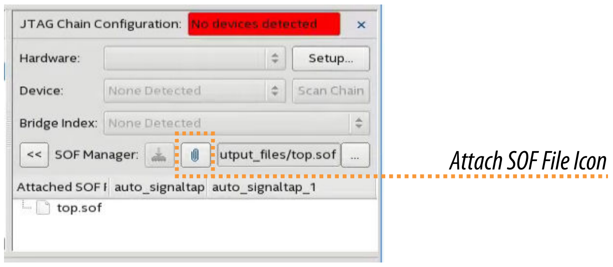
```
Data Log:
auto_signaltap_0
  signal_set: 2017/05/19 18:36:23 #0
    trigger: 2017/05/23 18:44:32 #0
      log: 2017/05/23 18:47:05 #0
  signal_set: 2017/05/23 18:50:55 #0
    trigger: 2017/05/23 18:50:55 #1
      log: 2017/05/23 18:51:05 #0
      log: 2017/05/23 18:51:05 #1
  signal_set: 2017/05/31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

14.3.10.2 SOF Manager

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

With the SOF Manager you can embed multiple SOFs into one `.stp` file. This action lets you move the `.stp` file to a different location, either on the same computer or across a network, without including the associated `.sof` separately. To embed a new SOF in the `.stp` file, click the **Attach SOF File** icon .

Figure 177. SOF Manager



JTAG Chain Configuration: No devices detected

Hardware:

Device: None Detected

Bridge Index: None Detected

SOF Manager: utput_files/top.sof

Attached SOF I	auto_signaltap	auto_signaltap_1
top.sof		

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

To download the new SOF to the FPGA, click the Program Device icon in the SOF Manager, after ensuring that the configuration of your `.stp` matches the design programmed into the target device.

Related Links

Data Log Pane on page 350

14.4 Defining Triggers

You specify various types of trigger conditions using the Signal Tap Logic Analyzer on the **Signal Configuration** pane. When you start the Signal Tap Logic Analyzer, it samples activity continuously from the monitored signals. The Signal Tap Logic Analyzer "triggers"—that is, the logic analyzer stops and displays the data—when a condition or set of conditions that you specified have been reached.

14.4.1 Basic Trigger Conditions

If you select the **Basic AND** or **Basic OR** trigger type, you must specify the trigger pattern for each signal you added in the `.stp`. To specify the trigger pattern, right-click the **Trigger Conditions** column and click the desired pattern. Set the trigger pattern to any of the following conditions:

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

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

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

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

Related Links

View, Analyze, and Use Captured Data on page 386

14.4.1.1 Using the Basic OR Trigger Condition with Nested Groups

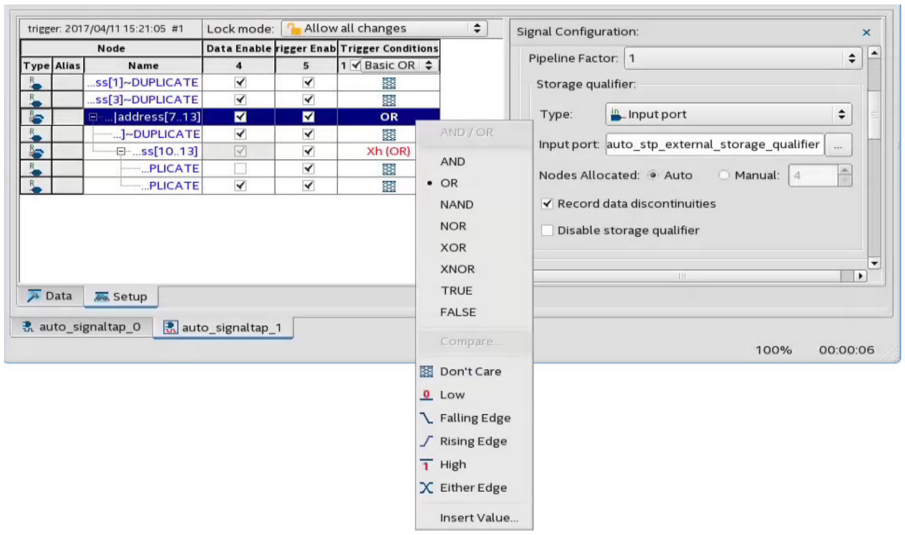
When you specify a set of signals as a nested group (group of groups) with the **Basic OR** trigger type, Signal Tap Logic Analyzer generates an advanced trigger condition. This condition sorts signals within groups to minimize the need to recompile your design. As long as the parent-child relationships of nodes are kept constant, the advanced trigger condition does not change. You can modify the sibling relationships of nodes and not need to recompile your design.

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

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

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

Figure 178. Applying Trigger Condition to Nested Group



14.4.2 Comparison Trigger Conditions

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

Signal Tap Logic Analyzer supports the following types of **Comparison** trigger conditions:

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

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

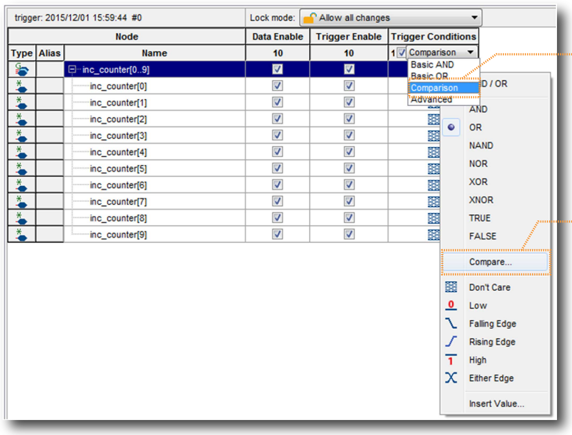
- Apply the **Comparison** trigger only to bus nodes consisting of leaf nodes.
- Do not form sub-groups within a bus node.
- Do not enable or disable individual trigger nodes within a bus node.
- Do not specify comparison values (in case of single-value comparison) or boundary values (in case of interval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

14.4.2.1 Specifying the Comparison Trigger Conditions

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

1. From the **Setup** tab, select **Comparison** under **Trigger Conditions**.
2. Right-click the node in the trigger editor, and select **Compare**.

Figure 179. Selecting the Comparison Trigger Condition



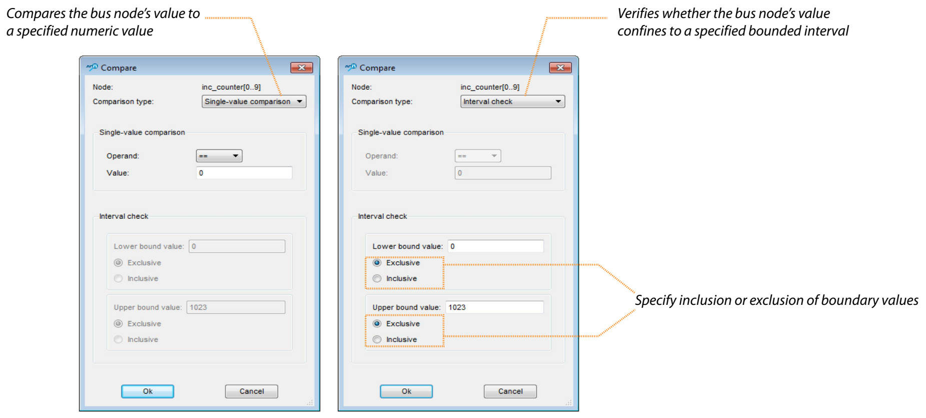
trigger: 2015/12/01 15:59:44 #0 | Lock mode: Allow all changes

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

3. Select the **Comparison type** from the Compare window.
   — If you choose **Single-value comparison** as your comparison type, specify the operand and value.
   — If you choose **Interval check** as your comparison type, provide the lower and upper bound values for the interval.

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

Figure 180. Specifying the Comparison Values


Compares the bus node's value to a specified numeric value
Verifies whether the bus node's value confines to a specified bounded interval
Specify inclusion or exclusion of boundary values
Compare
Ok
Cancel

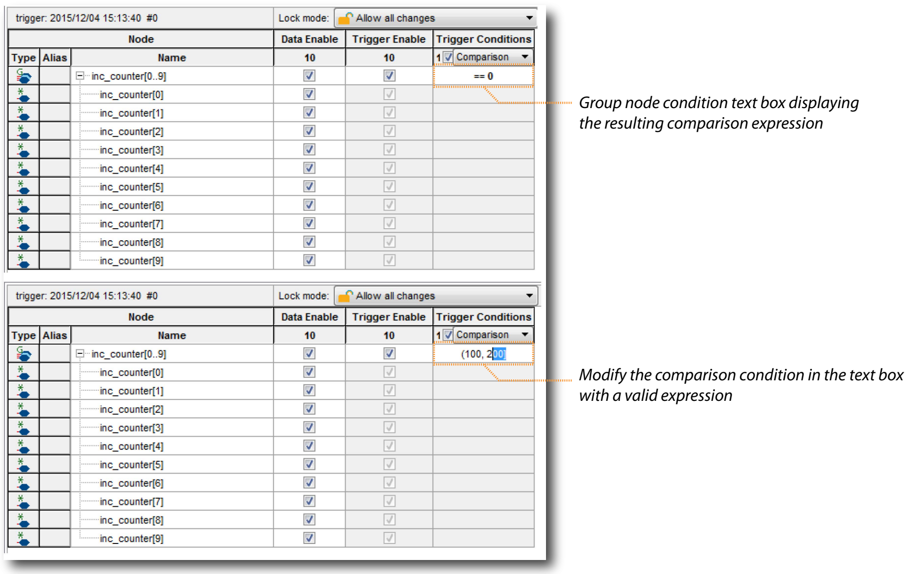
Compare
Node: inc_counter[0..9]
Comparison type: Single-value comparison
Single-value comparison
Operand: ==
Value: 0
Interval check
Lower bound value: 0
Exclusive
Inclusive
Upper bound value: 1023
Exclusive
Inclusive

Compare
Node: inc_counter[0..9]
Comparison type: Interval check
Single-value comparison
Operand: ==
Value: 0
Interval check
Lower bound value: 0
Exclusive
Inclusive
Upper bound value: 1023
Exclusive
Inclusive

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

   Note: You can modify the comparison condition in the text box with a valid expression.

Figure 181. Resulting Comparison Condition in Text Box


Group node condition text box displaying the resulting comparison expression
Modify the comparison condition in the text box with a valid expression

trigger: 2015/12/04 15:13:40 #0 Lock mode: Allow all changes

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

trigger: 2015/12/04 15:13:40 #0 Lock mode: Allow all changes

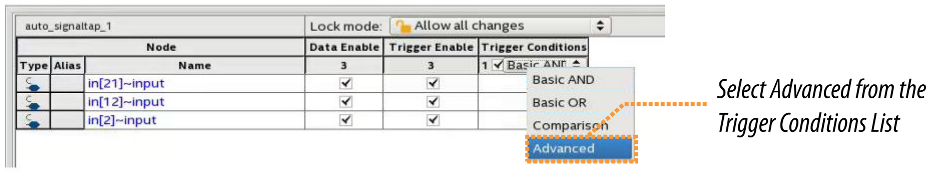
Node			Data Enable	Trigger Enable	Trigger Conditions
Type	Alias	Name	10	10	1 ✓ Comparison
		inc_counter[0..9]	✓	✓	(100, 200]
		inc_counter[0]	✓	✓	
		inc_counter[1]	✓	✓	
		inc_counter[2]	✓	✓	
		inc_counter[3]	✓	✓	
		inc_counter[4]	✓	✓	
		inc_counter[5]	✓	✓	
		inc_counter[6]	✓	✓	
		inc_counter[7]	✓	✓	
		inc_counter[8]	✓	✓	
		inc_counter[9]	✓	✓	

14.4.3 Advanced Trigger Conditions

To capture data for a given combination of conditions, build an advanced trigger. The Signal Tap Logic Analyzer provides the **Advanced Trigger** tab, which helps you build a complex trigger expression using a GUI.

Open the **Advanced Trigger** tab by selecting **Advanced** in the **Trigger Conditions** drop-down menu.

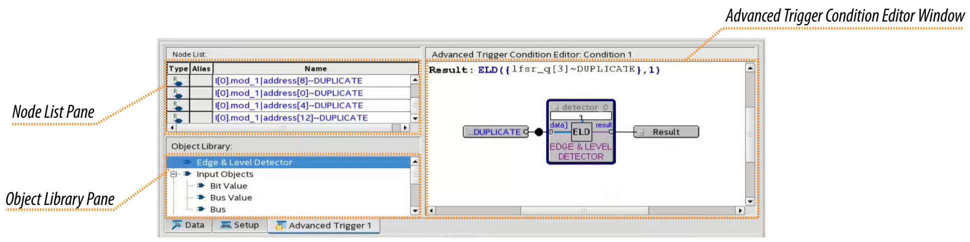
Figure 182. Accessing the Advanced Trigger Condition Tab



auto_signaltap_1 — Lock mode: Allow all changes

Type	Alias	Name	Data Enable	Trigger Enable	Trigger Conditions
			3	3	1 ✓ Basic AND
		in[21]~input	✓	✓	
		in[12]~input	✓	✓	
		in[2]~input	✓	✓	

Figure 183. Advanced Trigger Condition Tab



Node List:

Type	Alias	Name
		l[0].mod_1|address[8]~DUPLICATE
		l[0].mod_1|address[0]~DUPLICATE
		l[0].mod_1|address[4]~DUPLICATE
		l[0].mod_1|address[12]~DUPLICATE

Advanced Trigger Condition Editor: Condition 1

```
Result: ELD({lfsr_q[3]~DUPLICATE},1)
```

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

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

Table 111. Advanced Triggering Operators

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

continued...

Category	Name
	Right Shift
Custom Trigger HDL	

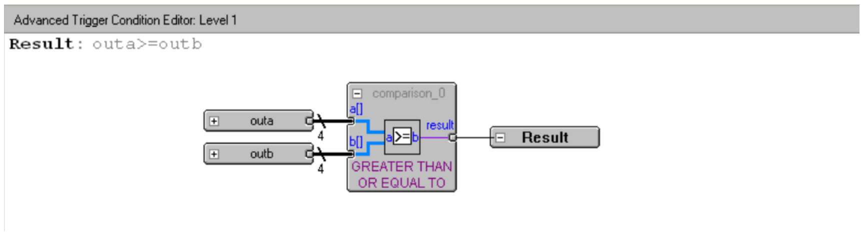
Adding many objects to the Advanced Trigger Condition Editor can make the work space cluttered and difficult to read. To keep objects organized while you build your advanced trigger condition, use the shortcut menu and select **Arrange All Objects**. Alternatively, use the **Zoom-Out** command to fit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

14.4.3.1 Examples of Advanced Triggering Expressions

The following examples show how to use Advanced Triggering:

Figure 184. Bus outa Is Greater Than or Equal to Bus outb

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


Advanced Trigger Condition Editor: Level 1

Result: outa>=outb

Figure 185. Enable Signal Has a Rising Edge

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

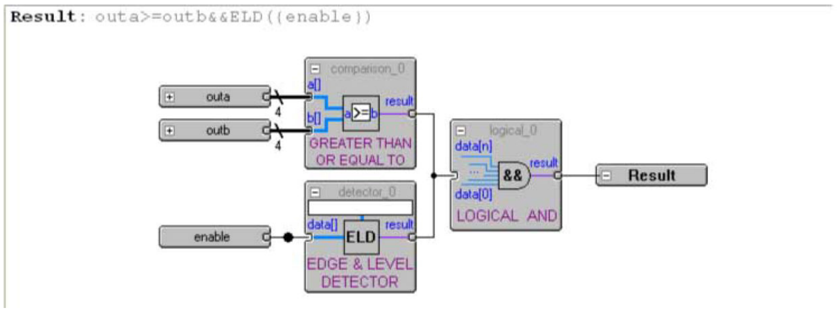
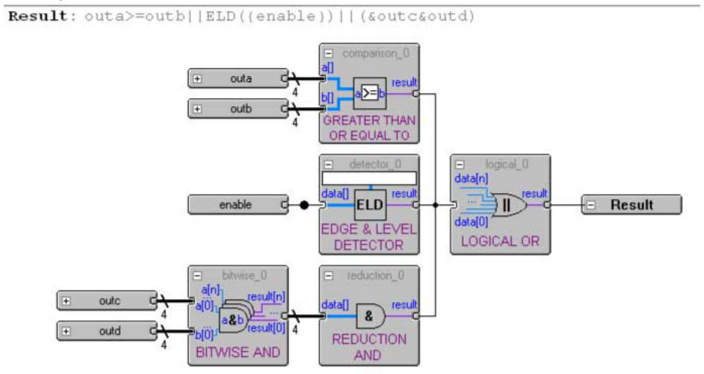

Result: outa>=outb&&ELD((enable))

Figure 186. Bitwise AND Operation

Trigger when bus outa is greater than or equal to bus outb, or when the enable signal has a rising edge. Or, when a bitwise AND operation has been performed between bus outc and bus outd, and all bits of the result of that operation are equal to 1.

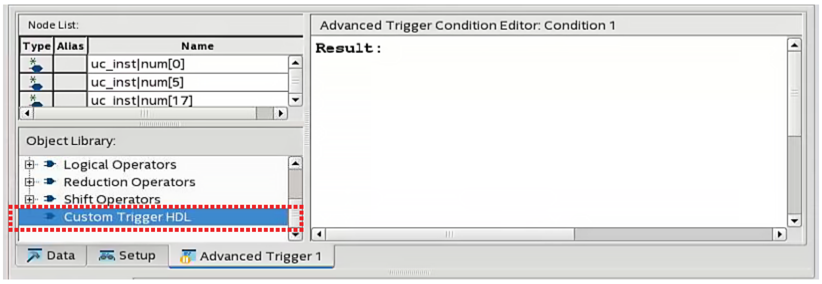

```
Result: outa>=outb||ELD((enable))||(&outc&outd)
```

14.4.4 Custom Trigger HDL Object

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

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

Figure 187. Object Library

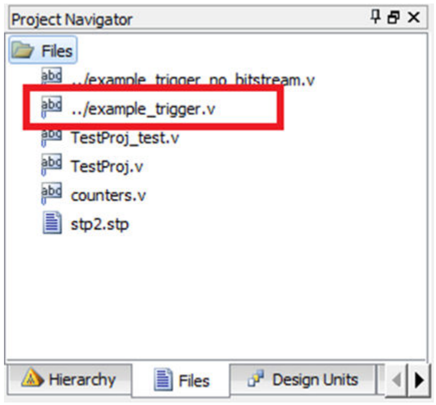


14.4.4.1 Using the Custom Trigger HDL Object

To define a custom trigger flow:

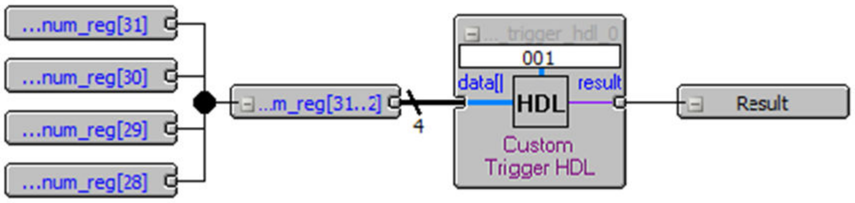
1. Select the trigger you want to edit.
2. Open the **Advanced Trigger** tab by selecting **Advanced** in the **Trigger Conditions** drop-down menu.
3. Add to your project the HDL source file that contains the trigger module using the **Project Navigator**.
   — Alternatively, append the HDL for your trigger module to a source file already included in the project.

Figure 188. HDL Trigger in the Project Navigator



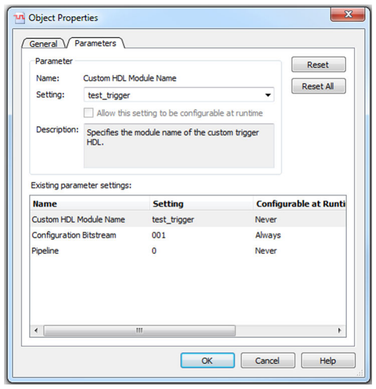
4. Implement the inputs and outputs that your Custom Trigger HDL module requires.
5. Drag in your Custom Trigger HDL object and connect the object's data input bus and result output bit to the final trigger result.

Figure 189. Custom Trigger HDL Object



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

Figure 190. Configure Object Properties



Parameter

Name: Custom HDL Module Name

Setting: test_trigger

☐ Allow this setting to be configurable at runtime

Description: Specifies the module name of the custom trigger HDL.

Existing parameter settings:

Name	Setting	Configurable at Runti
Custom HDL Module Name	test_trigger	Never
Configuration Bitstream	001	Always
Pipeline	0	Never

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

Example 35. Verilog HDL Triggers

The following trigger uses configuration bitstream:

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

This trigger does not have configuration bitstream:

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

14.4.4.2 Required Inputs and Outputs of Custom Trigger HDL Module

Table 112. Custom Trigger HDL Module Required Inputs and Outputs

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

14.4.4.3 Properties of Custom Trigger HDL Module

Table 113. Custom Trigger HDL Module Properties

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

14.4.5 Trigger Condition Flow Control

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

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

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

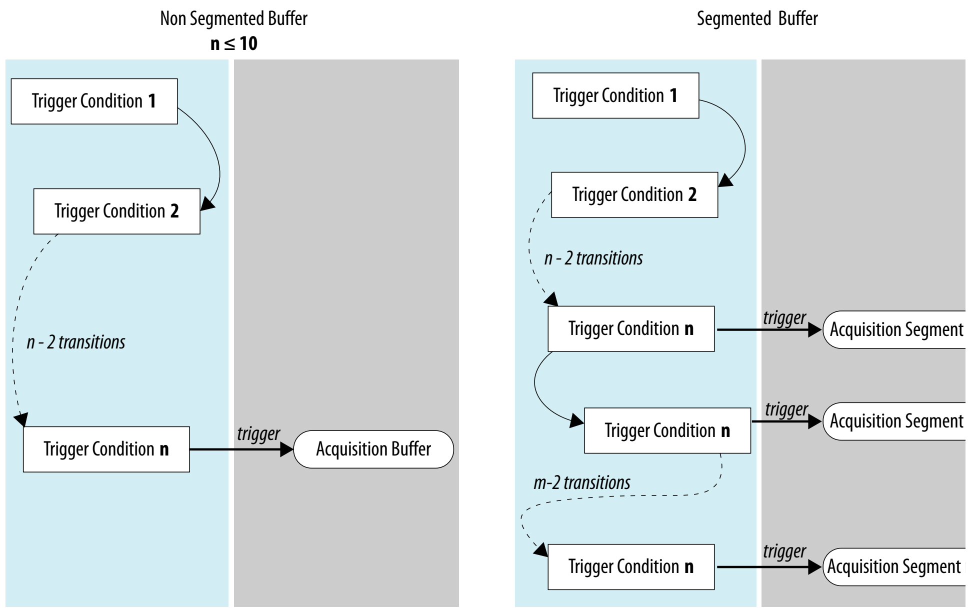
14.4.5.1 Sequential Triggering

Sequential triggering flow allows you to cascade up to 10 levels of triggering conditions. Signal Tap Logic Analyzer sequentially evaluates each of the conditions.

When the last triggering condition evaluates to TRUE, the Signal Tap Logic Analyzer triggers the acquisition buffer. For segmented buffers, every acquisition segment after the first triggers on the last condition that you specified. Use the Simple Sequential Triggering feature with basic triggers, comparison triggers, advanced triggers, or a mix of all three. The figure illustrates the simple sequential triggering flow for non-segmented and segmented buffers.

The external trigger is considered as trigger level `0`. The external trigger must be evaluated before the main trigger levels are evaluated.

Figure 191. Sequential Triggering Flow



Notes to figure:

1. The acquisition buffer starts capture when all n triggering levels are satisfied, where $n \leq 10$.
2. If you define an external trigger input, the Logic Analyzer evaluates it before evaluating all other trigger conditions.

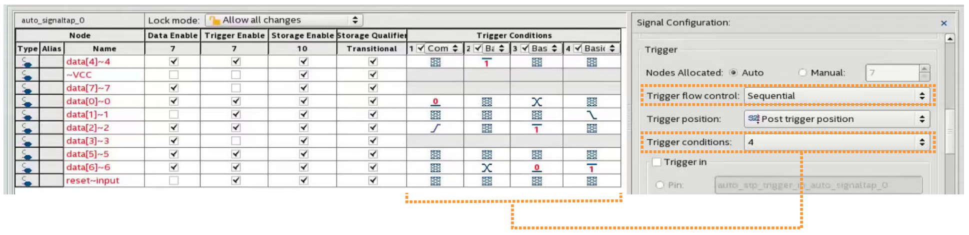
14.4.5.1.1 Configuring the Sequential Triggering Flow

To configure Signal Tap Logic Analyzer for sequential triggering:

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

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

Figure 192. Sequential Triggering Flow Configuration



auto_signaltap_0 Lock mode: Allow all changes

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

Signal Configuration:

Trigger

Nodes Allocated: [✓ Auto] Manual: 7

Trigger flow control: Sequential

Trigger position: Post trigger position

Trigger conditions: 4

☐ Trigger in

Pin: auto_stp_trigger_in_auto_signaltap_0

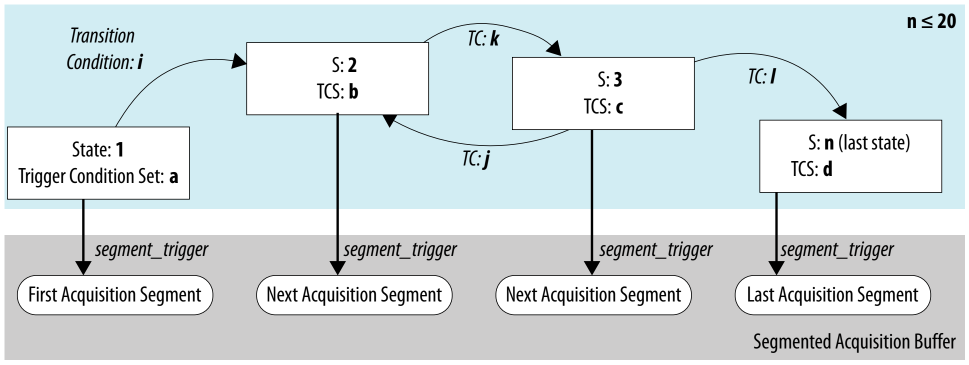
14.4.5.2 State-Based Triggering

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

Custom state-based triggering grants control over triggering condition arrangement, and allows for more efficient use of the space available in the acquisition buffer, because the Logic Analyzer only captures samples of interest.

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

Figure 193. State-Based Triggering Flow



Notes to figure:

1. You can define up to 20 different states.
2. If you define an external trigger input, the logic analyzer evaluates it before any conditions in the custom state-based triggering flow.

Each state allows you to define a set of conditional expressions. Each conditional expression is a Boolean expression that depends on a combination of triggering conditions, counters, and status flags. You configure the triggering conditions within the **Setup** tab. The Signal Tap Logic Analyzer custom-based triggering flow provides counters and status flags.

Within each conditional expression you define a set of actions. Actions include triggering the acquisition buffer to stop capture, a modification to either a counter or status flag, or a state transition.

Trigger actions can apply to either a single segment of a segmented acquisition buffer or to the entire non-segmented acquisition buffer. Each trigger action provides you with an optional count that specifies the number of samples the buffer captures before the logic analyzer stops acquisition of the current segment. The count argument allows you to control the amount of data the buffer captures before and after a triggering event occurs.

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

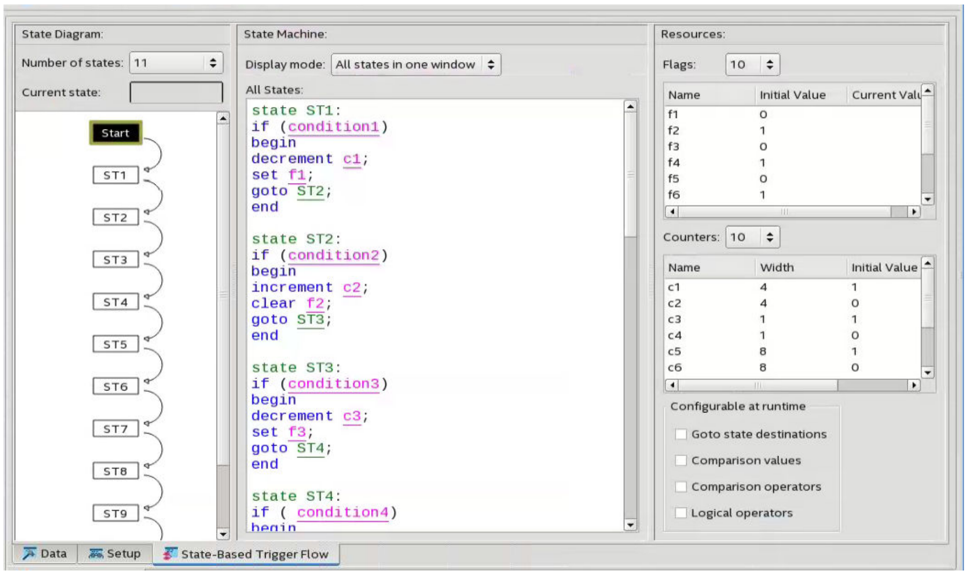
The state-based triggering flow allows you to capture a sequence of events that may not necessarily be contiguous in time. For example, a communication transaction between two devices that includes a hand shaking protocol containing a sequence of acknowledgements.

14.4.5.2.1 State-Based Triggering Flow Tab

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

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

Figure 194. State-Based Triggering Flow Tab



Number of states: 11

Current state:

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
```

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:

State Diagram Pane

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

State Machine Pane

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

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

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

Related Links

Signal Tap Trigger Flow Description Language on page 366

Resources Pane

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

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

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

Table 114. Runtime Reconfigurable Settings, State-Based Triggering Flow

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

Related Links

- Performance and Resource Considerations on page 380
- Runtime Reconfigurable Options on page 383

14.4.5.2.2 Trigger Lock Mode

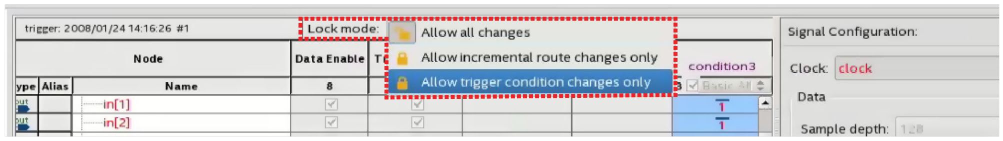
Trigger lock mode restricts changes to only the configuration settings that you specify as **Configurable at runtime**. The runtime configurable settings for the Custom Trigger Flow tab are on by default.

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

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

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

Figure 195. Allow Trigger Conditions Change Only



2. Modify the Trigger Flow conditions.

Incremental Route lock-mode restricts the GUI to only allow changes that require an Incremental Route compilation using Rapid Recompile. Use Rapid Recompile to perform incremental routing and gain a 2-4x speedup over the initial full compilation.

14.4.5.3 Signal Tap Trigger Flow Description Language

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

To describe the actions the Logic Analyzer evaluates when a state is reached, you follow this syntax:

Syntax of Trigger Flow Description Language

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

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

<state_label> on page 367

<boolean_expression> on page 367

<action_list> on page 368

Related Links

Custom Triggering Flow Application Examples on page 397

14.4.5.3.1 <state_label>

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

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

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

14.4.5.3.2 <boolean_expression>

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

Table 115. Logical Operators

Logical operators accept any boolean expression as an operand.

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

Table 116. Relational Operators

You use relational operators on counters or status flags.

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

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

Note:

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

14.4.5.3.3 <action_list>

List of actions that the Logic Analyzer performs within a state once a condition is satisfied.

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

Possible actions include:

Resource Manipulation Action

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

Table 117. Resource Manipulation Actions

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

Buffer Control Actions

Actions that control the acquisition buffer.

Table 118. Buffer Control Actions

Action	Description	Syntax
`trigger`	Stops the acquisition for the current buffer and ends analysis. This command is required in every flow definition.	`trigger <post-fill_count>;`
`segment_trigger`	Available only in segmented acquisition mode. Ends acquisition of the current segment. After evaluating this command, the Signal Tap Logic Analyzer starts acquiring from the next segment. If all segments are written, the Logic Analyzer	`segment_trigger <post-fill_count>;`

continued...

Action	Description	Syntax
	overwrites the oldest segment with the latest sample. When a trigger action is evaluated the acquisition stops.	
`start_store`	Active only in state-based storage qualifier mode. Asserts the `write_enable` to the Signal Tap acquisition buffer.	`start_store`
`stop_store`	Active only in state-based storage qualifier mode. De-asserts the `write_enable` signal to the Signal Tap acquisition buffer.	`stop_store`

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

Related Links

Post-fill Count on page 373

State Transition Action

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

14.4.5.4 Using the State-Based Storage Qualifier Feature

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

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

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

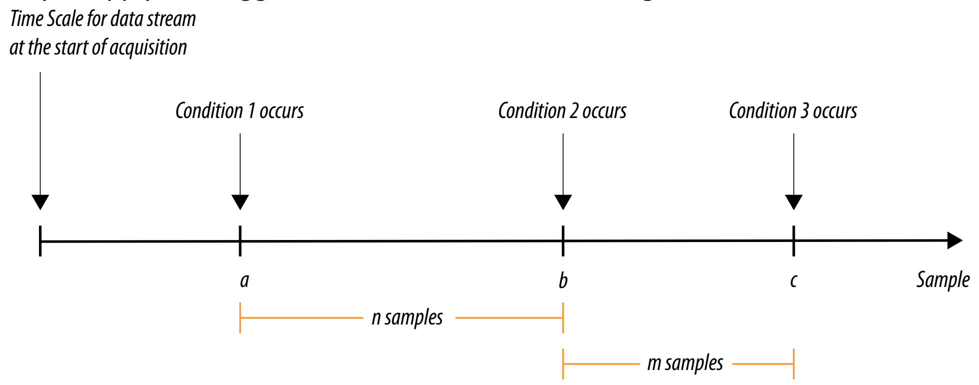
14.4.5.4.1 Storage Qualification Feature for the State-Based Trigger Flow.

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

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

Figure 196. Capture Scenario for Storage Qualification with the State-Based Trigger Flow

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



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

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

Example 36. Real data acquisition of the previous scenario

Figure 197. Storage Qualification with Post-Fill Count Value Less than m (Acquisition Successfully Completes)

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

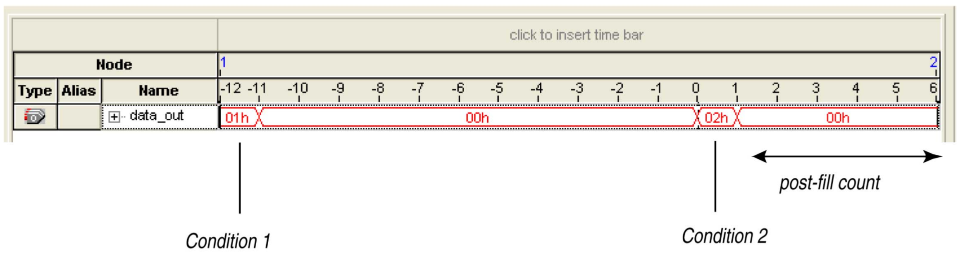
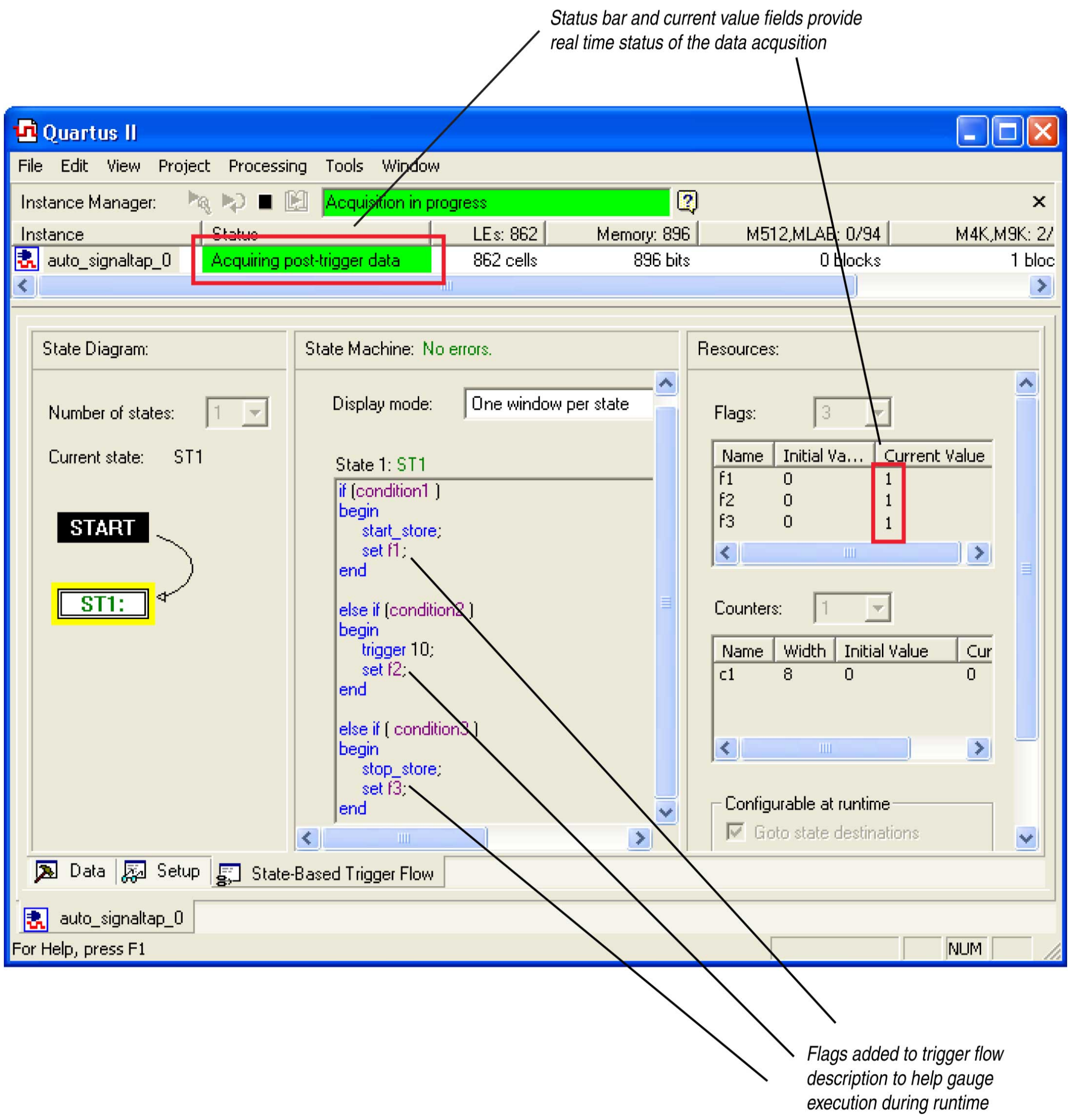


Figure 198. Storage Qualification with Post-Fill Count Value Greater than m (Acquisition Indefinitely Paused)

The logic analyzer pauses indefinitely, even after a trigger condition occurs due to a `stop_store` condition. This scenario uses a sample depth of 64, with `m = n = 10` and `post-fill count = 15`.

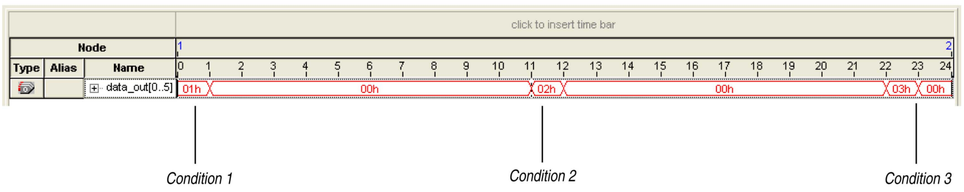


```
State 1: ST1
if (condition1 )
begin
    start_store;
    set f1;
end

else if (condition2 )
begin
    trigger 10;
    set f2;
end

else if ( condition3 )
begin
    stop_store;
    set f3;
end
```

Name	Initial Va...	Current Value
f1	0	1
f2	0	1
f3	0	1

Name	Width	Initial Value	Cur
c1	8	0	0

Figure 199. Waveform After Forcing the Analysis to Stop



The combination of using counters, Boolean and relational operators in conjunction with the `start_store` and `stop_store` commands can give a clock-cycle level of resolution to controlling the samples that are written into the acquisition buffer.

Example 37. Trigger flow description that skips three clock cycles of samples after hitting condition 1

Code:

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

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

Figure 200. Continuous Capture of Data Transaction

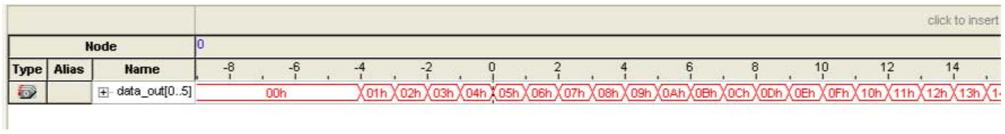
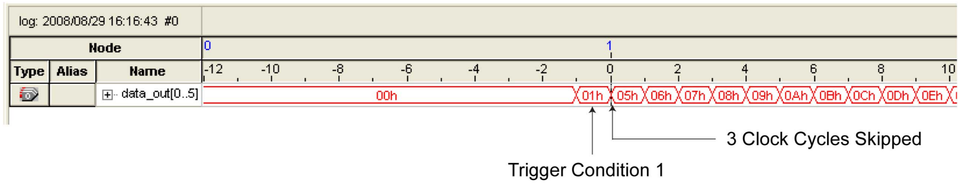


Figure 201. Capture of Data Transaction with Trigger Flow Description Applied



14.4.6 Specify Trigger Position

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

Signal Tap Logic Analyzer offers three pre-defined ratios of pre-trigger data to post-trigger data:

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

These pre-defined ratios apply to both non-segmented buffers and segmented buffers.

Related Links

State-Based Triggering on page 363

14.4.6.1 Post-fill Count

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

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

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

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

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

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

Related Links

Buffer Control Actions on page 368

14.4.7 Create a Power-Up Trigger

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

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

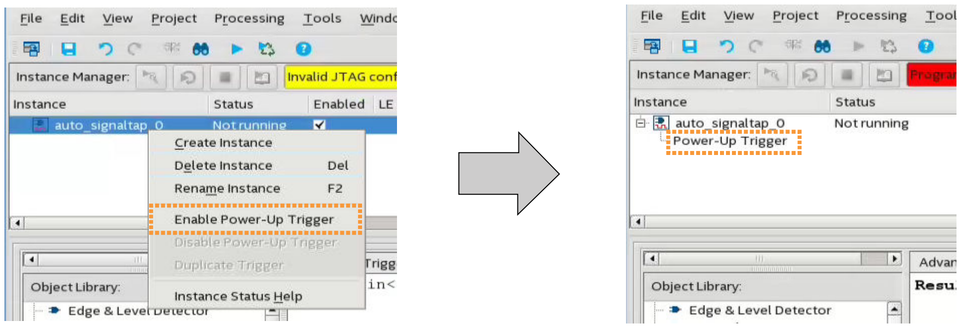
You can add a different Power-Up Trigger to each logic analyzer instance in the **Signal Tap Instance Manager** pane.

14.4.7.1 Enabling a Power-Up Trigger

To enable the Power-Up Trigger for a logic analyzer instance, right-click the instance and click **Enable Power-Up Trigger**. Alternatively, click **Edit ➤ Enable Power-Up Trigger**.

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

Figure 202. Enabling Power-Up Trigger in Signal Tap Logic Analyzer Editor



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

14.4.7.2 Manage and Configure Power-Up and Runtime Trigger Conditions

You can create basic, comparison, and advanced trigger conditions for your enabled Power-Up Trigger as you do with a Run-Time Trigger.

Since each instance now has two sets of trigger conditions—the Power-Up Trigger and the Run-Time Trigger—you can differentiate between the two with color coding. Power-Up Trigger conditions that you can adjust are color coded light blue, while Run-Time Trigger conditions you cannot adjust remain white.

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

You cannot make changes to Power-Up Trigger conditions that would normally require a full recompile with Runtime Trigger conditions, such as adding signals, deleting signals, or changing between basic, comparison, and advanced triggers. To apply these changes to the Power-Up Trigger conditions, first make the changes using the Runtime Trigger conditions.

Note: Any change made to the Power-Up Trigger conditions requires that you recompile the Signal Tap Logic Analyzer instance, even if a similar change to the Runtime Trigger conditions does not require a recompilation.

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

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

Related Links

Design Debugging Using In-System Sources and Probes on page 49

14.4.8 External Triggers

To trigger Signal Tap Logic Analyzer from an external source, you can create an external trigger input.

The external trigger input behaves like trigger condition 1, in that it must evaluate to TRUE before the logic analyzer evaluates any other configured trigger conditions.

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

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

- Use your processor debugger to configure the HPS to obey or disregard cross-trigger request from the FPGA, and to issue or not issue cross-trigger requests to the FPGA.
- Use your processor debugger in combination with the Signal Tap external trigger feature to develop a dynamic combination of cross-trigger behaviors.
- You can use the cross-triggering feature with the ARM Development Studio 5 (DS-5) software to implement a system-level debugging solution for your Intel FPGA SoC.

Related Links

- FPGA-Adaptive Software Debug and Performance Analysis white paper
- Signal Configuration Pane
  In *Intel Quartus Prime Help*

14.4.8.1 Using the Trigger Out of One Analyzer as the Trigger In of Another Analyzer

An advanced feature of the Signal Tap Logic Analyzer is the ability to use the **Trigger out** of one analyzer as the **Trigger in** to another analyzer. This feature allows you to synchronize and debug events that occur across multiple clock domains.

To perform this operation, first turn on **Trigger out** for the source logic analyzer instance. On the **Instance** list of the **Trigger out** trigger, select the targeted logic analyzer instance. For example, if the instance named `auto_signaltap_0` should trigger `auto_signaltap_1`, select `auto_signaltap_1|trigger_in` .

Turning on **Trigger out** automatically enables the **Trigger in** of the targeted logic analyzer instance and fills in the **Instance** field of the **Trigger in** trigger with the **Trigger out** signal from the source logic analyzer instance. In this example, `auto_signaltap_0` is targeting `auto_signaltap_1`. The Trigger In **Instance** field of `auto_signaltap_1` is automatically filled in with `auto_signaltap_0|trigger_out`.

14.5 Compiling the Design

To incorporate the Signal Tap logic in your design and enable the JTAG connection, you must compile your project. When you add a `.stp` file to your project, the Signal Tap Logic Analyzer becomes part of your design. When you debug your design with a traditional external logic analyzer, you must often make changes to the signals you want to monitor as well as the trigger conditions.

Note: Because these adjustments require that you recompile your design when using the Signal Tap Logic Analyzer, use the Signal Tap Logic Analyzer feature along with incremental compilation in the Intel Quartus Prime software to reduce recompilation time.

14.5.1 Faster Compilations with Intel Quartus Prime Incremental Compilation

You can add a Signal Tap Logic Analyzer instance to your design without recompiling your original source code. Incremental compilation enables you to preserve the synthesis and fitting results of your original design.

When you compile your design including a `.stp` file, Intel Quartus Prime software automatically adds the `sld_signaltap` and `sld_hub` entities to the compilation hierarchy. These two entities are the main components of the Signal Tap Logic Analyzer, providing the trigger logic and JTAG interface required for operation.

Incremental compilation is also useful when you want to modify the configuration of the `.stp` file. For example, you can change the buffer sample depth or memory type without performing a full compilation. Instead, you only recompile the Signal Tap Logic Analyzer, configured as its own design partition.

14.5.1.1 Enabling Incremental Compilation for Your Design

When enabled for your design, the Signal Tap Logic Analyzer is always a separate partition. After the first compilation, you can use the Signal Tap Logic Analyzer to analyze signals from the post-fit netlist. If your partitions are designed correctly, subsequent compilations due to Signal Tap Logic Analyzer settings take less time.

The netlist type for the top-level partition defaults to **source**. To take advantage of incremental compilation, specify the Netlist types for the partitions you want to tap as **Post-fit**.

Related Links

Intel Quartus Prime Incremental Compilation for Hierarchical and Team-Based Design documentation

14.5.1.2 Using Incremental Compilation with the Signal Tap Logic Analyzer

The Signal Tap Logic Analyzer uses the incremental compilation flow by default. For all signals that you want to connect to the Signal Tap Logic Analyzer from the post-fit netlist:

1. In the Design Partitions window, set the netlist type of the partition that contains the signals to **Post-Fit**, with a Fitter Preservation Level of **Placement and Routing**.
2. In the **Node Finder**, use the **Signal Tap: post-fitting filter** to add the signals of interest to your Signal Tap configuration file.
3. If you want to add signals from the pre-synthesis netlist, set the netlist type to **Source File** and use the **Signal Tap: pre-synthesis filter** in the **Node Finder**. Do not use the netlist type **Post-Synthesis** with the Signal Tap Logic Analyzer.

Caution: When using post-fit and pre-synthesis nodes:

- Read all incremental compilation guidelines to ensure the proper partitioning of a project.
- To speed up compile time, use only post-fit nodes for partitions specified as preservation-level post-fit.
- Do not mix pre-synthesis and post-fit nodes in any partition. If you must tap pre-synthesis nodes for a particular partition, make all tapped nodes in that partition pre-synthesis nodes and change the netlist type to **source** in the design partitions window.

Node names can differ between a pre-synthesis netlist and a post-fit netlist. In general, registers and user input signals share common names between the two netlists. During compilation, certain optimizations change the names of combinational signals in your RTL. If the type of node name chosen does not match the netlist type, the compiler may not be able to find the signal to connect to your Signal Tap Logic Analyzer instance for analysis. The compiler issues a critical warning to alert you of this scenario. The signal that is not connected is tied to ground in the **Signal Tap data** tab.

If you do use incremental compilation flow with the Signal Tap Logic Analyzer and source file changes are necessary, be aware that you may have to remove compiler-generated post-fit net names. Source code changes force the affected partition to go through resynthesis. During synthesis, the compiler cannot find compiler-generated net names from a previous compilation.

Note: Intel FPGA recommends using only registered and user-input signals as debugging taps in your `.stp` whenever possible.

Both registered and user-supplied input signals share common node names in the pre-synthesis and post-fit netlist. As a result, using only registered and user-supplied input signals in your `.stp` limits the changes you need to make to your Signal Tap Logic Analyzer configuration.

You can check the nodes that are connected to each Signal Tap instance using the In-System Debugging compilation reports. These reports list each node name you selected to connect to a Signal Tap instance, the netlist type used for the particular connection, and the actual node name used after compilation. If the incremental compilation flow is not used, the In-System Debugging reports are located in the Analysis & Synthesis folder. If the incremental compilation flow is used, this report is located in the Partition Merge folder.

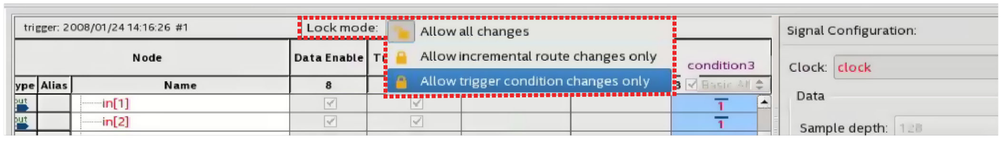
To verify that your original design was not modified, examine the messages in the **Partition Merge** section of the Compilation Report.

Unless you make changes to your design partitions that require recompilation, only the Signal Tap design partition is recompiled. If you make subsequent changes to only the `.stp`, only the Signal Tap design partition must be recompiled, reducing your recompilation time.

14.5.2 Prevent Changes Requiring Recompilation

Configure the `.stp` to prevent changes that normally require recompilation. To do this, select a **Lock mode** from above the node list in the **Setup** tab. To lock your configuration, choose **Allow trigger condition changes only**.

Figure 203. Allow Trigger Conditions Change Only



Related Links

Verify Whether You Need to Recompile Your Project on page 382

14.5.3 Incremental Route with Rapid Recompile

You can use Incremental Route with Rapid Recompile to decrease compilation times. After performing a full compilation on your design, you can use the Incremental Route flow to achieve a 2-4x speedup over a flat compile. The Incremental Route flow is not compatible with Partial Reconfiguration.

Intel Quartus Prime Standard Edition software supports Incremental Route with Rapid Recompile for Arria V, Cyclone V, and Stratix V devices.

Related Links

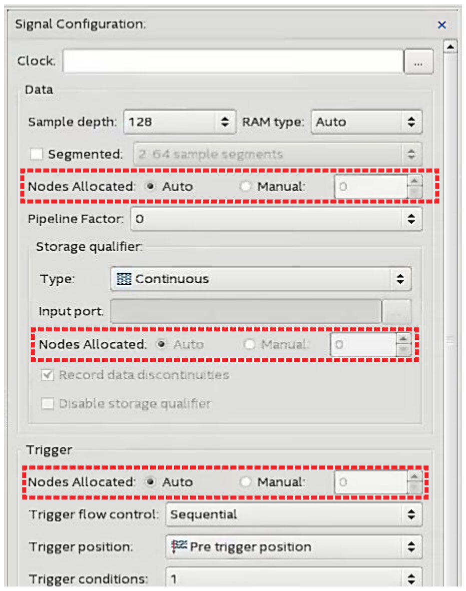
Running Rapid Recompile
In *Intel Quartus Prime Pro Edition Handbook Volume 1*

14.5.3.1 Using the Incremental Route Flow

To use the Incremental Route flow:

1. Open your design and run **Analysis & Elaboration** (or a full compilation) to give node visibility in Signal Tap.
2. Add Signal Tap to your design.
3. In the Signal Tap **Signal Configuration** pane, specify **Manual** in the **Nodes Allocated** field for Trigger and Data nodes (and Storage Qualifier, if used).

Figure 204. Manually Allocate Nodes



Signal Configuration:

Clock:

Data

Sample depth: 128 RAM type: Auto

Segmented: 2 64 sample segments

Nodes Allocated: [✓ Auto] Manual: 0

Pipeline Factor: 0

Storage qualifier:

Type: Continuous

Input port:

Nodes Allocated: [✓ Auto] Manual: 0

[✓] Record data discontinuities

Disable storage qualifier

Trigger

Nodes Allocated: [✓ Auto] Manual: 0

Trigger flow control: Sequential

Trigger position: Pre trigger position

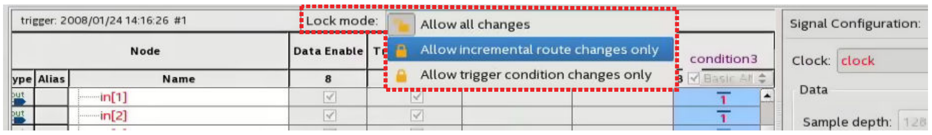
Trigger conditions: 1

Manual node allocation allows you to control the number of nodes compiled into the design, which is critical for the Incremental Route flow.

When you select **Auto** allocation, the number of nodes compiled into the design matches the number of nodes in the **Setup** tab. If you add a node later, you create a mismatch between the amount of nodes the device requires and the amount of compiled nodes, and you must perform a full compilation.

4. Specify the number of nodes that you estimate necessary for the debugging process. You can increase the number of nodes later, but this requires more compilation time.
5. Add the nodes that you want to tap.
6. If you have not fully compiled your project, run a full compilation. Otherwise, start incremental compile using Rapid Recompile.
7. Debug and determine additional signals of interest.
8. (Optional) Select **Allow incremental route changes only** lock-mode.

Figure 205. Incremental Route Lock-Mode



trigger: 2008/01/24 14:16:26 #1

Lock mode:
- Allow all changes
- [✓ Allow incremental route changes only]
- Allow trigger condition changes only

Type	Alias	Name	Data Enable 8	Trigger Enable	condition3
		in[1]	✓	✓	
		in[2]	✓	✓	

Signal Configuration:

Clock: clock

Data

Sample depth: 128

9. Add additional nodes in the Signal Tap **Setup** tab.
   — Do not exceed the number of manually allocated nodes you specified.
   — Avoid making changes to non-runtime configurable settings.
10. Click the Rapid Recompile icon from the toolbar. Alternatively, click **Processing ➤ Start Rapid Recompile**.

Note: The previous steps set up your design for Incremental Route, but the actual Incremental Route process begins when you perform a Rapid Recompile.

14.5.3.2 Tips to Achieve Maximum Speedup

- Basic AND (which applies to Storage Qualifier as well as trigger input) is the fastest for the Incremental Route flow.
- Basic OR is slower for the Incremental Route flow, but if you avoid changing the parent-child relationship of nodes within groups, you can minimize the impact on compile time. You can change the sibling relationships of nodes.
  — Basic OR and advanced triggers require re-synthesis when you change the number/names of tapped nodes.
- Use the Incremental Route lock-mode to avoid inadvertent changes requiring a full compilation.

14.5.4 Timing Preservation with the Signal Tap Logic Analyzer

In addition to verifying functionality, timing closure is one of the most crucial processes in successful operation of your design.

Note: When you compile a project with a Signal Tap Logic Analyzer without the use of incremental compilation, you must add IP to your existing design. This addition often impacts the existing placement, routing, and timing of your design. To minimize the effect that the Signal Tap Logic Analyzer has on your design, use incremental compilation for your project. Incremental compilation is the default setting in new designs. You can easily enable incremental compilation in existing designs. When the Signal Tap Logic Analyzer is in a design partition, it has little to no affect on your design.

For Intel Arria 10 devices, the Intel Quartus Prime Standard Edition software does not support timing preservation for post-fit taps with Rapid Recompile.

Use the following techniques to help maintain timing:

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

Related Links

Timing Closure and Optimization
In *Intel Quartus Prime Standard Edition Handbook Volume 2*

14.5.5 Performance and Resource Considerations

When you perform logic analysis of your design, you can see the necessary trade-off between runtime flexibility, timing performance, and resource usage.

The Signal Tap Logic Analyzer allows you to select runtime configurable parameters to balance the need for runtime flexibility, speed, and area.

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

14.5.5.1 Signal Tap Logic in Critical Path

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

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

14.5.5.2 Signal Tap Logic Using Critical Resources

If your design is resource constrained, follow these tips to reduce the logic or memory the Signal Tap Logic Analyzer uses:

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

14.6 Program the Target Device or Devices

After you add the Signal Tap Logic Analyzer to your project and re-compile, you can configure the FPGA target device.

If you want to debug multiple designs simultaneously, configure the device from the `.stp` instead of the Intel Quartus Prime Programmer. This allows you to open more than one `.stp` file and program multiple devices.

14.6.1 Ensure Setting Compatibility Between .stp and .sof Files

A `.stp` file is compatible with a `.sof` file when the settings for the logic analyzer, such as the size of the capture buffer and the signals you use for monitoring or triggering, match the programming settings of the target device. If the files are not compatible, you can still program the device, but you cannot run or control the logic analyzer from the Signal Tap Logic Analyzer Editor.

- To ensure programming compatibility, program your device with the `.sof` file generated in the most recent compilation.
- To check whether a particular `.sof` is compatible with the current Signal Tap configuration, attach the `.sof` to the SOF manager.

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

Although having a Intel Quartus Prime project is not required when using an `.stp`, it is recommended. The project database contains information about the integrity of the current Signal Tap Logic Analyzer session. Without the project database, there is no way to verify that the current `.stp` file matches the `.sof` file in the device. If you have an `.stp` file that does not match the `.sof` file, the Signal Tap Logic Analyzer can capture incorrect data.

Related Links

Manage Multiple Signal Tap Files and Configurations on page 350

14.6.2 Verify Whether You Need to Recompile Your Project

Before starting a debugging session, do not make any changes to the `.stp` settings that require recompiling the project.

To verify whether a change you made requires recompiling the project, check the Signal Tap status display at the top of the **Instance Manager** pane. This feature allows you to undo the change, so that you do not need to recompile your project.

Related Links

Prevent Changes Requiring Recompilation on page 378

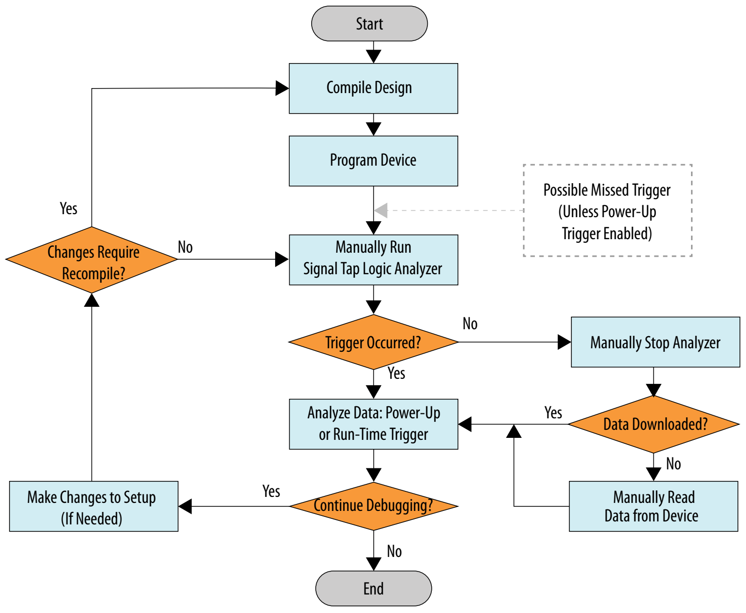
14.7 Running the Signal Tap Logic Analyzer

Debugging Signal Tap Logic Analyzer is similar using an external logic analyzer. You initialize the logic analyzer by starting an analysis. When your trigger event occurs, the logic analyzer stores the captured data in the device's memory buffer, and then transfers this data to the `.stp` file with the JTAG connection.

You can also perform the equivalent of a force trigger instruction that lets you view the captured data currently in the buffer without a trigger event occurring.

The flowchart shows how you operate the Signal Tap Logic Analyzer. indicates where Power-Up and Runtime Trigger events occur and when captured data from these events is available for analysis.

Figure 206. Power-Up and Runtime Trigger Events Flowchart


Start
Compile Design
Program Device
Possible Missed Trigger (Unless Power-Up Trigger Enabled)
Yes
Changes Require Recompile?
No
Manually Run Signal Tap Logic Analyzer
Trigger Occurred?
No
Manually Stop Analyzer
Yes
Analyze Data: Power-Up or Run-Time Trigger
Yes
Data Downloaded?
No
Make Changes to Setup (If Needed)
Yes
Continue Debugging?
Manually Read Data from Device
No
End

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

Related Links

Design Debugging Using In-System Sources and Probes on page 49

14.7.1 Runtime Reconfigurable Options

When you use Runtime Trigger mode, you can change certain settings in the `.stp` without recompiling your design.

Table 119. Runtime Reconfigurable Features

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

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

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

In Incremental Route lock mode, **Allow incremental route changes only**, limits to changes that only require an Incremental Route compilation, and not a full compile.

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

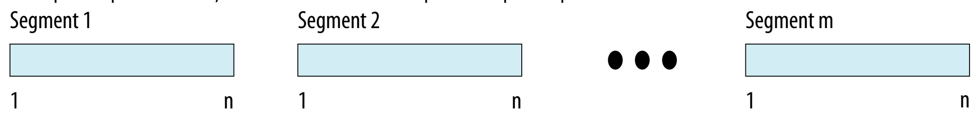
```
state ST1:
if ( condition1 && (c1 <= m) )// each "segment"  triggers on condition
                              // 1
begin                           // m  = number of total "segments"
   start_store;
   increment c1;
   goto ST2:
End

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

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

else (c2 < n)
begin
    increment c2;
    goto ST2;
end
```

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

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

Figure 207. Segmented Buffer Created with Storage Qualifier and State-Based Trigger

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



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

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

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

```
state ST1 :
    if (condition2  && f1)              // additional state added for a non-
segmented
                                       // acquisition set f1 to enable state
        begin
             start_store;
             trigger
        end
    else if (! f1)
        goto ST2;
state ST2:
    if ( (condition1 && (c1 <= m)  && f2) // f2 status flag used to mask
state. Set f2
                                         // to enable
        begin
           start_store;
           increment c1;
           goto ST3:
        end
    else (c1 > m )
           start_store
   Trigger (n-1)
    end
state ST3:
    if ( c2 >= n)
        begin
            reset c2;
            stop_store;
            goto ST1;
        end
    else (c2 < n)
    begin
        increment c2;
        goto ST2;
    end
```

14.7.2 Signal Tap Status Messages

The table describes the text messages that might appear in the Signal Tap Status Indicator in the **Instance Manager** pane before, during, and after a data acquisition. Use these messages to monitor the state of the logic analyzer or what operation it is performing.

Table 120. Text Messages in the Signal Tap Status Indicator

Message	Message Description
`Not running`	The Signal Tap Logic Analyzer is not running. There is no connection to a device or the device is not configured.
`(Power-Up Trigger) Waiting for clock (1)`	The Signal Tap Logic Analyzer is performing a Runtime or Power-Up Trigger acquisition and is waiting for the clock signal to transition.

continued...

Message	Message Description
`Acquiring (Power-Up) pre-trigger data` (1)	The trigger condition has not been evaluated yet. A full buffer of data is collected if using the non-segmented buffer acquisition mode and storage qualifier type is continuous.
`Trigger In conditions met`	Trigger In condition has occurred. The Signal Tap Logic Analyzer is waiting for the condition of the first trigger condition to occur. This can appear if Trigger In is specified.
`Waiting for (Power-up) trigger` (1)	The Signal Tap Logic Analyzer is now waiting for the trigger event to occur.
`Trigger level` <x> met	The condition of trigger condition *x* has occurred. The Signal Tap Logic Analyzer is waiting for the condition specified in condition x + 1 to occur.
`Acquiring (power-up) post-trigger data` (1)	The entire trigger event has occurred. The Signal Tap Logic Analyzer is acquiring the post-trigger data. The amount of post-trigger data collected is you define between 12%, 50%, and 88% when the non-segmented buffer acquisition mode is selected.
`Offload acquired (Power-Up) data` (1)	Data is being transmitted to the Intel Quartus Prime software through the JTAG chain.
`Ready to acquire`	The Signal Tap Logic Analyzer is waiting for you to initialize the analyzer.

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

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

14.8 View, Analyze, and Use Captured Data

Use the Signal Tap Logic Analyzer interface to examine the data you captured manually or using a trigger, and use your findings to debug your design.

When in the Data view, you can use the drag-to-zoom feature by left-clicking to isolate the data of interest.

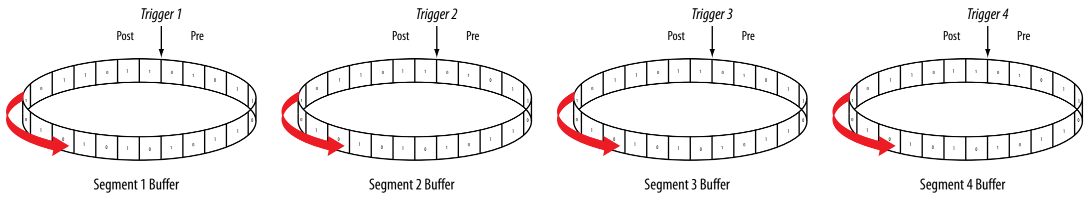
14.8.1 Capturing Data Using Segmented Buffers

Segmented Acquisition buffers allow you to perform multiple captures with a separate trigger condition for each acquisition segment. This feature allows you to capture a recurring event or sequence of events that span over a long period time efficiently.

Each acquisition segment acts as a non-segmented buffer, continuously capturing data when it is activated. When you run an analysis with the **segmented buffer** option enabled, the Signal Tap Logic Analyzer performs back-to-back data captures for each acquisition segment within your data buffer. You define the trigger flow, or the type and order in which the trigger conditions evaluate for each buffer, either in the Sequential trigger flow control or in the Custom State-based trigger flow control.

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

Figure 208. Segmented Acquisition Buffer

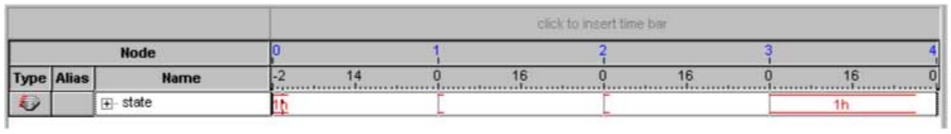


The Signal Tap Logic Analyzer finishes an acquisition with a segment, and advances to the next segment to start a new acquisition. Depending on when a trigger condition occurs, it may affect the way the data capture appears in the waveform viewer. The figure illustrates the data capture method. The Trigger markers—Trigger 1, Trigger 2, Trigger 3 and Trigger 4—refer to the evaluation of the `segment_trigger` and `trigger` commands in the Custom State-based trigger flow. If you use a sequential flow, the Trigger markers refer to trigger conditions specified within the **Setup** tab.

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

The next figure shows an example of a capture using sequential flow control with the trigger condition for each segment specified as **Don't Care**.

Figure 209. Segmented Capture with Preemption of Acquisition Segments



Each segment before the last captures only one sample, because the next trigger condition immediately preempts capture of the current buffer. The trigger position for all segments is specified as pre-trigger (10% of the data is before the trigger condition and 90% of the data is after the trigger position). Because the last segment starts immediately with the trigger condition, the segment contains only post-trigger data. The three empty samples in the last segment are left over from the pre-trigger samples that the Signal Tap Logic Analyzer allocated to the buffer.

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

14.8.2 Differences in Pre-fill Write Behavior Between Different Acquisition Modes

The Signal Tap Logic Analyzer uses one of the following three modes when writing into the acquisition memory:

- **Non-segmented buffer**
- **Non-segmented buffer with a storage qualifier**
- **Segmented buffer**

There are subtle differences in the amount of data captured immediately after running the Signal Tap Logic Analyzer and before any trigger conditions occur. A non-segmented buffer, running in continuous mode, completely fills the buffer with sampled data before evaluating any trigger conditions. Thus, a non-segmented capture without any storage qualification enabled always shows a waveform with a full buffer's worth of data captured.

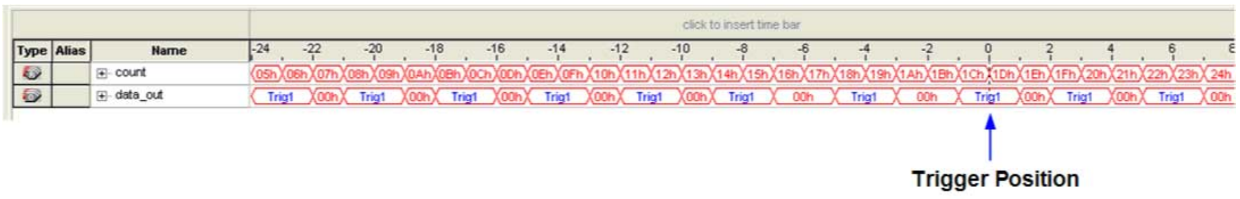
Filling the buffer provides you with as much data as possible within the capture window. The buffer gets pre-filled with data samples prior to evaluating the trigger condition. As such, Signal Tap requires that the buffer be filled at least once before any data can be retrieved through the JTAG connection and prevents the buffer from being dumped during the first acquisition prior to a trigger condition when you perform a **Stop Analysis**.

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

If the trigger event occurs on any data sample before the specified amount of pre-trigger data has occurred, then the Signal Tap Logic Analyzer triggers and begins filling memory with post-trigger data, regardless of the amount of pre-trigger data you specify. For example, if you set the trigger position to 50% and set the logic analyzer to trigger on a processor reset, start the logic analyzer, and then power on your target system, the logic analyzer triggers. However, the logic analyzer memory is filled only with post-trigger data, and not any pre-trigger data, because the trigger event, which has higher precedence than the capture of pre-trigger data, occurred before the pre-trigger condition was satisfied.

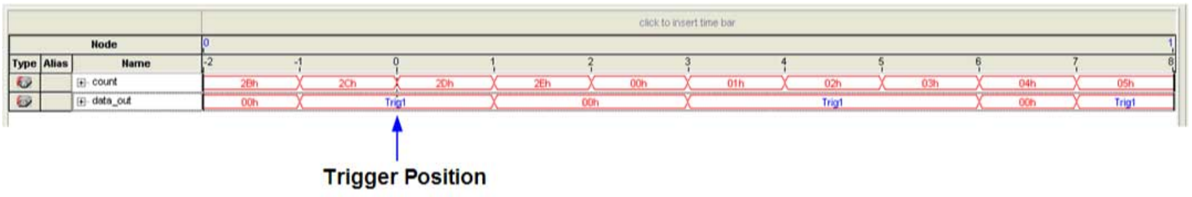
The figures for continuous data capture and conditional data capture show the difference between a non-segmented buffer in continuous mode and a non-segmented buffer using a storage qualifier. The configuration of the logic analyzer waveforms below is a base trigger condition, sample depth of 64 bits,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

Figure 210. Signal Tap Logic Analyzer Continuous Data Capture


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

In the continuous data capture, Trig1 occurs several times in the data buffer before the Signal Tap Logic Analyzer actually triggers. A full buffer's worth of data is captured before the logic analyzer evaluates any trigger conditions. After the trigger condition occurs, the logic analyzer continues acquisition until it captures eight additional samples (12% of the buffer, as defined by the "post-trigger" position).

Figure 211. Signal Tap Logic Analyzer Conditional Data Capture



Note to figure:

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

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

14.8.3 Creating Mnemonics for Bit Patterns

The mnemonic table feature allows you to assign a meaningful name to a set of bit patterns, such as a bus. To create a mnemonic table, right-click in the **Setup** or **Data** tab of an `.stp` and click **Mnemonic Table Setup**. You create a mnemonic table by entering sets of bit patterns and specifying a label to represent each pattern. Once you have created a mnemonic table, assign the table to a group of signals. To assign a mnemonic table, right-click on the group, click **Bus Display Format** and select the desired mnemonic table.

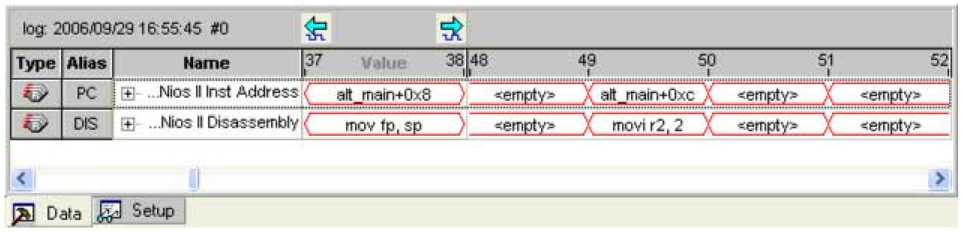
You use the labels you create in a table in different ways on the **Setup** and **Data** tabs. On the **Setup** tab, you can create basic triggers with meaningful names by right-clicking an entry in the **Trigger Conditions** column and selecting a label from the table you assigned to the signal group. On the **Data** tab, if any captured data matches a bit pattern contained in an assigned mnemonic table, the signal group data is replaced with the appropriate label, making it easy to see when expected data patterns occur.

14.8.4 Automatic Mnemonics with a Plug-In

When you use a plug-in to add signals to an `.stp`, mnemonic tables for the added signals are automatically created and assigned to the signals defined in the plug-in. To enable these mnemonic tables manually, right-click the name of the signal or signal group. On the **Bus Display Format** shortcut menu, then click the name of the mnemonic table that matches the plug-in.

As an example, the Nios II plug-in helps you to monitor signal activity for your design as the code is executed. If you set up the logic analyzer to trigger on a function name in your Nios II code based on data from an `.elf`, you can see the function name in the **Instance Address** signal group at the trigger sample, along with the corresponding disassembled code in the **Disassembly** signal group, as shown in Figure 13–52. Captured data samples around the trigger are referenced as offset addresses from the trigger function name.

Figure 212. Data Tab when the Nios II Plug-In is Used



14.8.5 Locating a Node in the Design

When you find the source of an error in your design using the Signal Tap Logic Analyzer, you can use the node locate feature to locate that signal in many of the tools found in the Intel Quartus Prime software, as well as in your design files. This lets you find the source of the problem quickly so you can modify your design to correct the flaw. To locate a signal from the Signal Tap Logic Analyzer in one of the Intel Quartus Prime software tools or your design files, right-click the signal in the `.stp`, and click **Locate in <tool name>**.

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

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

14.8.6 Saving Captured Data

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

When analysis is set to **Auto-run mode**, the Logic Analyzer creates a separate entry in the Data Log to store the data captured each time the trigger occurred. This allows you to review the captured data for each trigger event.

The default name for a log is based time stamp when the Logic Analyzer acquired the data. As a best practice, rename the data log with a more meaningful name.

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

Related Links

Data Log Pane on page 350

14.8.7 Exporting Captured Data to Other File Formats

You can export captured data to the following file formats, for use with other EDA simulation tools:

- Comma Separated Values File (`.csv`)
- Table File (`.tbl`)
- Value Change Dump File (`.vcd`)
- Vector Waveform File (`.vwf`)
- Graphics format files (`.jpg`, `.bmp`)

To export the captured data from Signal Tap Logic Analyzer, on the File menu, click **Export** and specify the **File Name**, **Export Format**, and **Clock Period**.

14.8.8 Creating a Signal Tap List File

A `.stp` list file contains all the data the logic analyzer captures for a trigger event, in text format.

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

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

To create a `.stp` list file in the Intel Quartus Prime software, click **File ➤ Create/ Update ➤ Create Signal Tap List File**.

Related Links

Adding Signals with a Plug-In on page 337

14.9 Other Features

The Signal Tap Logic Analyzer provides optional features not specific to a task flow. The following techniques may offer advantages in specific circumstances.

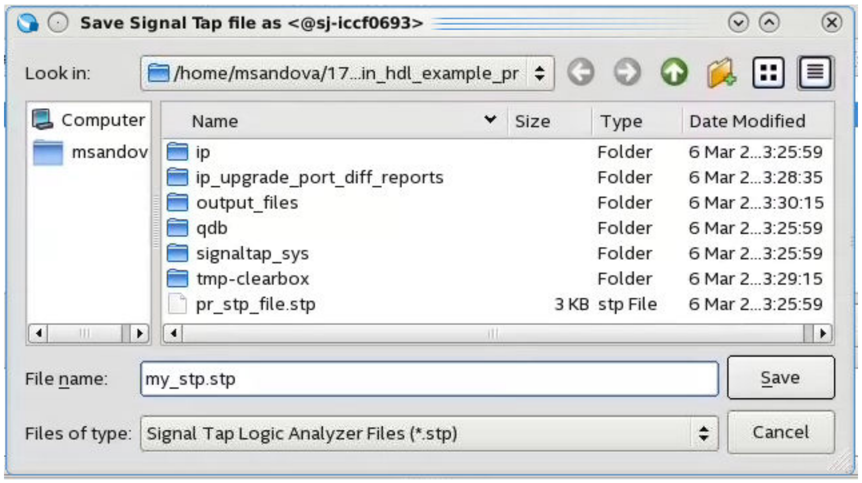
14.9.1 Creating Signal Tap File from Design Instances

In addition to providing GUI support for generation of `.stp` files, the Intel Quartus Prime software supports generation of a Signal Tap instance from logic defined in HDL source files. This technique is helpful to modify runtime configurable trigger conditions, acquire data, and view acquired data on the data log via Signal Tap utilities.

14.9.1.1 Creating a .stp File from a Design Instance

To generate a `.stp` file from parameterized HDL instances within your design:

1. Open or create an Intel Quartus Prime project that includes one or more HDL instances of the Signal Tap logic analyzer.
2. Click **Processing ➤ Start ➤ Start Analysis & Synthesis**.
3. Click **File ➤ Create/Update ➤ Create Signal Tap File from Design Instance(s)**.
4. Specify a location for the `.stp` file that generates, and click **Save**.


Save Signal Tap file as <@sj-iccf0693>

Look in: /home/msandova/17...in_hdl_example_pr

Computer
msandov

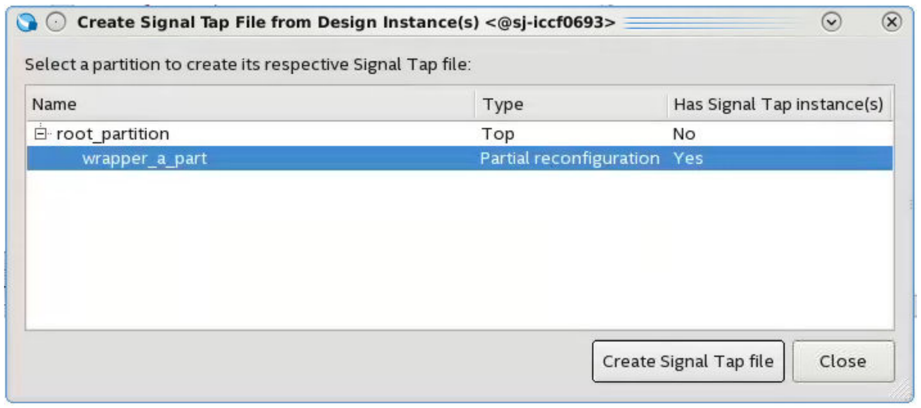
Name	Size	Type	Date Modified
ip		Folder	6 Mar 2...3:25:59
ip_upgrade_port_diff_reports		Folder	6 Mar 2...3:28:35
output_files		Folder	6 Mar 2...3:30:15
qdb		Folder	6 Mar 2...3:25:59
signaltap_sys		Folder	6 Mar 2...3:25:59
tmp-clearbox		Folder	6 Mar 2...3:29:15
pr_stp_file.stp	3 KB	stp File	6 Mar 2...3:25:59

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

Figure 213. Create Signal Tap File from Design Instances Dialog Box

Note: If your project contains partial reconfiguration partitions, the **Create Signal Tap File from Design Instance(s)** dialog box displays a tree view of the PR partitions in the project. Select a partition from the view, and click **Create Signal Tap file**. The resultant `.stp` file that generates contains all HDL instances in the corresponding PR partition. The resultant `.stp` file does not include the instances in any nested partial reconfiguration partition.

Figure 214. Selecting Partition for `.stp` File Generation

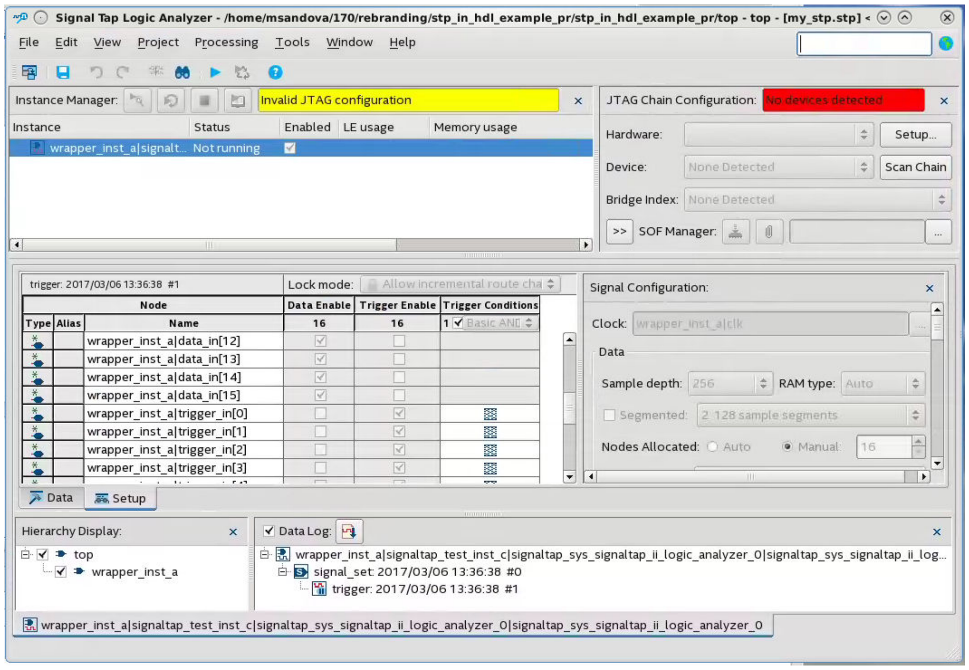

Create Signal Tap File from Design Instance(s) <@sj-iccf0693>

Select a partition to create its respective Signal Tap file:

Name	Type	Has Signal Tap instance(s)
root_partition	Top	No
wrapper_a_part	Partial reconfiguration	Yes

After successful `.stp` file creation, the **Signal Tap Logic Analyzer** appears. All the fields are read-only, except runtime-configurable trigger conditions.

Figure 215. Generated `.stp` File



Signal Tap Logic Analyzer - /home/msandova/170/rebranding/stp_in_hdl_example_pr/stp_in_hdl_example_pr/top - top - [my_stp.stp]

Instance Manager: Invalid JTAG configuration

Instance	Status	Enabled	LE usage	Memory usage
wrapper_inst_a|signalt...	Not running	✓		

JTAG Chain Configuration: No devices detected

Hardware:

Device: None Detected

Bridge Index: None Detected

SOF Manager:

trigger: 2017/03/06 13:36:38 #1

Lock mode: Allow incremental route cha

Node			Data Enable	Trigger Enable	Trigger Conditions
Type	Alias	Name	16	16	1 ✓ Basic AND
		wrapper_inst_a|data_in[12]	✓		
		wrapper_inst_a|data_in[13]	✓		
		wrapper_inst_a|data_in[14]	✓		
		wrapper_inst_a|data_in[15]	✓		
		wrapper_inst_a|trigger_in[0]		✓	▨
		wrapper_inst_a|trigger_in[1]		✓	▨
		wrapper_inst_a|trigger_in[2]		✓	▨
		wrapper_inst_a|trigger_in[3]		✓	▨

Signal Configuration:

Clock: wrapper_inst_a|clk

Data

Sample depth: 256 RAM type: Auto

Segmented: 2 128 sample segments

Nodes Allocated: Auto [✓ Manual] 16

Hierarchy Display:

- ✓ top
  - ✓ wrapper_inst_a

✓ Data Log:

- wrapper_inst_a|signaltap_test_inst_c|signaltap_sys_signaltap_ii_logic_analyzer_0|signaltap_sys_signaltap_ii_log...
  - signal_set: 2017/03/06 13:36:38 #0
    - trigger: 2017/03/06 13:36:38 #1

wrapper_inst_a|signaltap_test_inst_c|signaltap_sys_signaltap_ii_logic_analyzer_0|signaltap_sys_signaltap_ii_logic_analyzer_0

Related Links

- Create Signal Tap File from Design Instances
  In *Intel Quartus Prime Help*
- Custom Trigger HDL Object on page 358

14.9.2 Using the Signal Tap MATLAB MEX Function to Capture Data

When you use MATLAB for DSP design, you can acquire data from the Signal Tap Logic Analyzer directly into a matrix in the MATLAB environment by calling the MATLAB MEX function `alt_signaltap_run`, built into the Intel Quartus Prime software. If you use the MATLAB MEX function in a loop, you can perform as many acquisitions in the same amount of time as you can when using Signal Tap in the Intel Quartus Prime software environment.

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

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

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

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

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

3. Save the `.stp` file and compile your design. Program your device and run the Signal Tap Logic Analyzer to ensure your trigger conditions and signal acquisition work correctly.

4. In the MATLAB environment, add the Intel Quartus Prime binary directory to your path with the following command:

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

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

   `alt_signaltap_run`

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

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

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

   When capturing data, you must assign a filename, for example, *<stp filename>* as a requirement of the MATLAB MEX function. Other MATLAB MEX function options are described in the table:

Table 121. Signal Tap MATLAB MEX Function Options

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

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

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

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

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

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

14.9.3 Using Signal Tap in a Lab Environment

You can install a stand-alone version of the Signal Tap Logic Analyzer. This version is particularly useful in a lab environment in which you do not have a workstation that meets the requirements for a complete Intel Quartus Prime installation, or if you do not have a license for a full installation of the Intel Quartus Prime software. The standalone version of the Signal Tap Logic Analyzer is included with and requires the Intel Quartus Prime stand-alone Programmer which is available from the Downloads page of the Altera website.

14.9.4 Remote Debugging Using the Signal Tap Logic Analyzer

14.9.4.1 Debugging Using a Local PC and an SoC

You can use the System Console with Signal Tap Logic Analyzer to remote debug your Intel FPGA SoC. This method requires one local PC, an existing TCP/IP connection, a programming device at the remote location, and an Intel FPGA SoC.

Related Links

Remote Hardware Debugging over TCP/IP

14.9.4.2 Debugging Using a Local PC and a Remote PC

You can use the Signal Tap Logic Analyzer to debug a design that is running on a device attached to a PC in a remote location.

To perform a remote debugging session, you must have the following setup:

- The Intel Quartus Prime software installed on the local PC
- Stand-alone Signal Tap Logic Analyzer or the full version of the Intel Quartus Prime software installed on the remote PC
- Programming hardware connected to the device on the PCB at the remote location
- TCP/IP protocol connection

14.9.4.2.1 Equipment Setup

1. On the PC in the remote location, install the standalone version of the Signal Tap Logic Analyzer, included in the Intel Quartus Prime stand-alone Programmer, or the full version of the Intel Quartus Prime software.
2. Connect the remote computer to Intel programming hardware, such as the or Intel FPGA Download Cable.
3. On the local PC, install the full version of the Intel Quartus Prime software.
4. Connect the local PC to the remote PC across a LAN with the TCP/IP protocol.

14.9.5 Using the Signal Tap Logic Analyzer in Devices with Configuration Bitstream Security

Certain device families support bitstream decryption during configuration using an on-device AES decryption engine. You can still use the Signal Tap Logic Analyzer to analyze functional data within the FPGA. However, note that JTAG configuration is not possible after the security key has been programmed into the device.

Intel FPGA recommends that you use an unencrypted bitstream during the prototype and debugging phases of the design. Using an unencrypted bitstream allows you to generate new programming files and reconfigure the device over the JTAG connection during the debugging cycle.

If you must use the Signal Tap Logic Analyzer with an encrypted bitstream, first configure the device with an encrypted configuration file using Passive Serial (PS), Fast Passive Parallel (FPP), or Active Serial (AS) configuration modes. The design must contain at least one instance of the Signal Tap Logic Analyzer. After the FPGA is configured with a Signal Tap Logic Analyzer instance in the design, when you open the Signal Tap Logic Analyzer in the Intel Quartus Prime software, you then scan the chain and are ready to acquire data with the JTAG connection.

14.9.6 Monitor FPGA Resources Used by the Signal Tap Logic Analyzer

The Signal Tap Logic Analyzer has a built-in resource estimator that calculates the logic resources and amount of memory that each logic analyzer instance uses. Furthermore, because the most demanding on-chip resource for the logic analyzer is memory usage, the resource estimator reports the ratio of total RAM usage in your design to the total amount of RAM available, given the results of the last compilation. The resource estimator provides a warning if a potential for a “no-fit” occurs.

You can see resource usage (by instance and total) in the columns of the **Instance Manager** pane of the Signal Tap Logic Analyzer Editor. Use this feature when you know that your design is running low on resources.

The logic element value that the resource usage estimator reports may vary by as much as 10% from the actual resource usage.

14.10 Design Example: Using Signal Tap Logic Analyzers

The system in this example contains many components, including a Nios processor, a direct memory access (DMA) controller, on-chip memory, and an interface to external SDRAM memory. After you press a button, the processor initiates a DMA transfer, which you analyze using the Signal Tap Logic Analyzer. In this example, the Nios processor executes a simple C program from on-chip memory and waits for you to press a button.

Related Links

AN 446: Debugging Nios II Systems with the Signal Tap Embedded Logic Analyzer application note

14.11 Custom Triggering Flow Application Examples

The custom triggering flow in the Signal Tap Logic Analyzer is most useful for organizing a number of triggering conditions and for precise control over the acquisition buffer. This section provides two application examples for defining a custom triggering flow within the Signal Tap Logic Analyzer. Both examples can be easily copied and pasted directly into the state machine description box by using the state display mode **All states in one window**.

Related Links

On-chip Debugging Design Examples website

14.11.1 Design Example 1: Specifying a Custom Trigger Position

Actions to the acquisition buffer can accept an optional post-count argument. This post-count argument enables you to define a custom triggering position for each segment in the acquisition buffer.

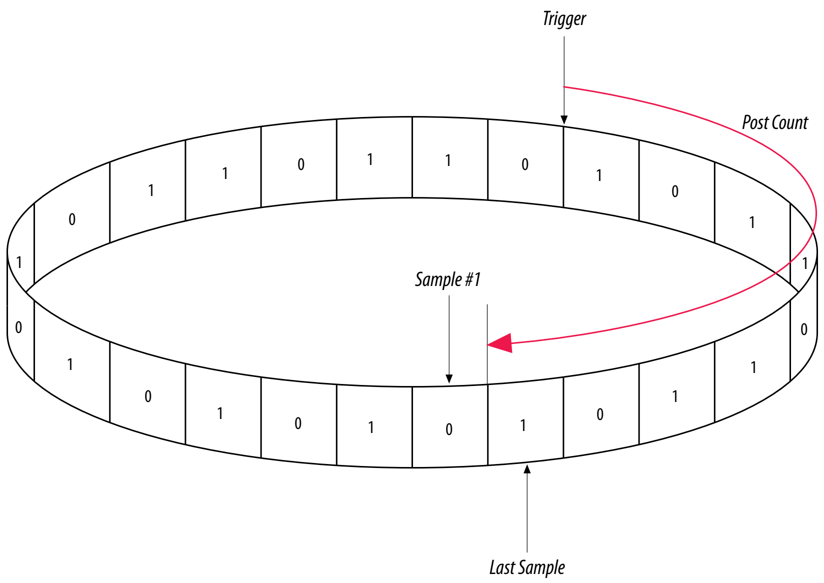
The example shows how to apply a trigger position to all segments in the acquisition buffer. The example describes a triggering flow for an acquisition buffer split into four segments. If each acquisition segment is 64 samples in depth, the trigger position for each buffer will be at sample #34. The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

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

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

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

Figure 216. Specifying a Custom Trigger Position



14.11.2 Design Example 2: Trigger When triggercond1 Occurs Ten Times between triggercond2 and triggercond3

The custom trigger flow description is often useful to count a sequence of events before triggering the acquisition buffer. The example shows such a sample flow. This example uses three basic triggering conditions configured in the Signal Tap **Setup** tab.

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

```
state ST1:
if ( condition2  )
begin
   reset c1;
   goto ST2;
end
State ST2 :
if ( condition1 )
    increment c1;
else if (condition3 && c1 < 10)
    goto ST3;
else if ( condition3 && c1 >= 10)
    trigger;
ST3:
goto ST3;
```

14.12 Signal Tap Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Intel Quartus Prime Command-Line and Tcl API Help browser. To run the Help browser, type the following at the command prompt:

```
quartus_sh --qhelp
```

Related Links

Tcl Scripting
In *Intel Quartus Prime Standard Edition Handbook Volume 2*

14.12.1 Signal Tap Tcl Commands

The `quartus_stp` executable supports a Tcl interface that allows you to capture data without running the Intel Quartus Prime GUI. You cannot execute Signal Tap Tcl commands from within the Tcl console in the Intel Quartus Prime software. You must run them from the command-line with the `quartus_stp` executable. To execute a Tcl file that has Signal Tap Logic Analyzer Tcl commands, use the following command:

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

Example 38. Continuously capturing data

This excerpt shows commands you can use to continuously capture data. Once the capture meets trigger condition e, the data is captured and stored in the data log.

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

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

#  Calling run_multiple_end will start all instances
run_multiple_start

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

run_multiple_end

# Close Signal Tap session
close_session
```

Related Links

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

14.12.2 Signal Tap Command-Line Options

To compile your design with the Signal Tap Logic Analyzer using the command prompt, use the `quartus_stp` command. You can use the following options with the `quartus_stp` executable:

Table 122. `quartus_stp` Command-Line Options

Option	Usage	Description
`--stp_file` *<stp_filename>*	Mandatory	Specifies the name of the `.stp` file.
`--enable`	Optional	Creates assignments to the specified `.stp` in the `.qsf` and changes ENABLE_SIGNALTAP to ON. Includes Signal Tap Logic Analyzer in the next compilation. If no `.stp` is specified in the `.qsf`, the `--stp_file` option must be used. If omitted, the compiler uses the current value of ENABLE_SIGNALTAP in the `.qsf file` .
`--disable`	Optional	Removes the `.stp` reference from the `.qsf` and changes ENABLE_SIGNALTAP to OFF. The Signal Tap Logic Analyzer is removed from the design database the next time you compile your design. If the `--disable` option is omitted, the current value of ENABLE_SIGNALTAP in the `.qsf` is used.
`--create_signaltap_hdl_file`	Optional	Creates an `.stp` representing the Signal Tap instance. You must use the `--stp_file` option to create an `.stp`. Equivalent to the **Create Signal Tap File from Design Instance(s)** command in the Intel Quartus Prime software.

The first example illustrates how to compile a design with the Signal Tap Logic Analyzer at the command line.

```
quartus_stp filtref --stp_file stp1.stp --enable
quartus_map filtref --source=filtref.bdf --family=CYCLONE
quartus_fit filtref --part=EP1C12Q240C6 --fmax=80MHz --tsu=8ns
quartus_asm filtref
```

The `quartus_stp --stp_file stp1.stp --enable` command creates the QSF variable and instructs the Intel Quartus Prime software to compile the `stp1.stp` file with your design. The `--enable` option must be applied for the Signal Tap Logic Analyzer to compile into your design.

The example below shows how to create a new `.stp` after building the Signal Tap Logic Analyzer instance with the IP Catalog.

```
quartus_stp filtref --create_signaltap_hdl_file --stp_file stp1.stp
```

Related Links

Command-Line Scripting
In *Intel Quartus Prime Standard Edition Handbook Volume 2*

14.13 Document Revision History

Table 123. Document Revision History

Date	Version	Changes Made
2017.11.06	17.1.0	• Clarified information about the Data Log Pane. • Updated Figure: Data Log and renamed to Simple Data Log. • Added Figure: Accessing the Advanced Trigger Condition Tab.
2017.05.08	17.0.0	• Added: Open Standalone Signal Tap Logic Analyzer GUI. • Updated figures on Create Signal Tap File from Design Instance(s).
2016.10.31	16.1.0	• Added: Create Signal Tap File from Design Instance(s). • Removed reference to unsupported Talkback feature.
2016.05.03	16.0.0	• Added: Specifying the Pipeline Factor • Added: Comparison Trigger Conditions
2015.11.02	15.1.0	Changed instances of *Quartus II* to *Intel Quartus Prime*.
2015.05.04	15.0.0	Added content for Floating Point Display Format in table: Signal Tap Logic Analyzer Features and Benefits.
2014.12.15	14.1.0	Updated location of Fitter Settings, Analysis & Synthesis Settings, and Physical Synthesis Optimizations to Compiler Settings.
December 2014	14.1.0	• Added MAX 10 as supported device. • Removed Full Incremental Compilation setting and Post-Fit (Strict) netlist type setting information. • Removed outdated GUI images from "Using Incremental Compilation with the Signal Tap Logic Analyzer" section.
June 2014	14.0.0	• DITA conversion. • Replaced MegaWizard Plug-In Manager and Megafunction content with IP Catalog and parameter editor content. • Added flows for custom trigger HDL object, Incremental Route with Rapid Recompile, and nested groups with Basic OR. • GUI changes: toolbar, drag to zoom, disable/enable instance, trigger log time-stamping.
November 2013	13.1.0	Removed HardCopy material. Added section on using cross-triggering with DS-5 tool and added link to white paper 01198. Added section on remote debugging an Altera SoC and added link to application note 693. Updated support for MEX function.
May 2013	13.0.0	• Added recommendation to use the state-based flow for segmented buffers with separate trigger conditions, information about Basic OR trigger condition, and hard processor system (HPS) external triggers. • Updated "Segmented Buffer" on page 13-17, Conditional Mode on page 13-21, Creating Basic Trigger Conditions on page 13-16, and Using External Triggers on page 13-48.
June 2012	12.0.0	Updated Figure 13–5 on page 13–16 and "Adding Signals to the Signal Tap File" on page 13–10.
November 2011	11.0.1	Template update. Minor editorial updates.
May 2011	11.0.0	Updated the requirement for the standalone Signal Tap software.
December 2010	10.0.1	Changed to new document template.

continued...

Date	Version	Changes Made
July 2010	10.0.0	• Add new acquisition buffer content to the "View, Analyze, and Use Captured Data" section. • Added script sample for generating hexadecimal CRC values in programmed devices. • Created cross references to Intel Quartus Prime Help for duplicated procedural content.
November 2009	9.1.0	No change to content.
March 2009	9.0.0	• Updated Table 13–1 • Updated "Using Incremental Compilation with the Signal Tap Logic Analyzer" on page 13–45 • Added new Figure 13–33 • Made minor editorial updates
November 2008	8.1.0	Updated for the Intel Quartus Prime software version 8.1 release: • Added new section "Using the Storage Qualifier Feature" on page 14–25 • Added description of `start_store` and `stop_store` commands in section "Trigger Condition Flow Control" on page 14–36 • Added new section "Runtime Reconfigurable Options" on page 14–63
May 2008	8.0.0	Updated for the Intel Quartus Prime software version 8.0: • Added "Debugging Finite State machines" on page 14-24 • Documented various GUI usability enhancements, including improvements to the resource estimator, the bus find feature, and the dynamic display updates to the counter and flag resources in the State-based trigger flow control tab • Added "Capturing Data Using Segmented Buffers" on page 14–16 • Added hyperlinks to referenced documents throughout the chapter • Minor editorial updates

Related Links

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